AFRL-IF-RS-TR-2002-236
Final Technical Report
September 2002

JOINT FORCE AIR COMPONENT COMMANDER (JFACC) ACTIVE TECHNOLOGY

BBN Technologies

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. E951

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2002-236 has been reviewed and is approved for publication.

APPROVED:

MARK D. FORESTI
Project Engineer

FOR THE DIRECTOR:

MICHAEL L. TALBERT, Maj., USAF
Technical Advisor
Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE SEPTEMBER 2002	3. REPORT TYPE AND DATES COVERED Final Apr 97 – Apr 01

4. TITLE AND SUBTITLE
JOINT FORCE AIR COMPONENT COMMANDER (JFACC) ACTIVE TECHNOLOGY

5. FUNDING NUMBERS
C - F30602-97-C-0075
PE - 62301E
PR - E951
TA - 01
WU - 00

6. AUTHOR(S)
Maurice McNeil, Susan Banks, Paul Neves, Bernice Allison, David Perham, Joe Kraska, and Gauri Sukhatankar

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
BBN Technologies
9655 Granite Ridge Drive
Suite 245
San Diego California 92123

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency   AFRL/IFTD
3701 North Fairfax Drive   525 Brooks Road
Arlington Virginia 22203-1714   Rome New York 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2002-236

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Mark Foresti/IFTD/(315) 330-2233/ Mark.Foresti@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 Words)*
The objective of this effort is to design, develop & implement active database technologies for addressing critical issues of providing timely, accurate data to the JFACC after next planning & execution process. This effort is applicable to multiple sub-areas of the technology base research & development category: managing the planning & execution process and flows of information in the process, continuous replanning & managing plan changes.

14. SUBJECT TERMS
Active Database Technology, Sentinels, Monitors, Database Techniques

15. NUMBER OF PAGES
171

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

TABLE OF CONTENTS

List of Figures

List of Tables

Overview

1   Abstract

This project consisted of two phases. Phase I was a Research Base effort under DARPA's JFACC program. During Phase I we implemented active database technologies, originally developed by the University of Florida[1], as a stand-alone Event-Condition-Action (ECA) Service within the futuristic JFACC architecture. The ECA Service could interact with multiple data sources and applications across either LANs or WANs. The Phase I demonstration showed the integration of current and forecast weather from the Weather Anchor Desk with the DARPA JFACC Plan Service and associated applications.

Phase II was sponsored by the CACC program. Phase II demonstrated the ECA Service with real-world Air Force, Navy, WWW, and DLA data sources and models. Data sources included TBMCS, DLA's ICIS logistics data and models and NOTAM data from the WWW. An interface to the Navy's TEDS weather service was also prototyped, but not integrated. Alerts were generated and displayed in a variety of interfaces including email, browsers, a stand-alone alert interface, and on wireless devices. Phase II included the implementation of a Local Event Detector (LED) integrated with an Oracle™ 8.1 RDBMS. The LED utilized the Oracle trigger mechanisms without significant impact on the performance of the database.

---

[1] AFRL-IF-RS-TR-1999-6 "Distributed Events in Sentinel: Design and Implementation of a Global Event Detector", Chakravarthy, Liao, Kim, January 1999

2   Introduction and Background

The objective of this project is to transition and apply active database technologies from laboratory research to providing timely, accurate data to the Air Force Air Campaign planning and execution process.

Air Force operations are currently data centric when they need to be knowledge centric. In order to support continuous, "just-in-time" planning and execution, the Air Force needs to provide planners with accurate, timely, and relevant knowledge. It is not enough for planners to be advised that an aberration in the schedule has occurred. Planners must comprehend: What has happened? How has it effected ongoing operations? How might it effect ongoing operations?, How will it effect the current planning cycle? In the context of joint operations, the impact of these changes is much broader than just the ATO planning cycle. Ground, Air, Sea and Special operations are tightly choreographed to maximize combat power involving complex interdependencies and critical timing. The impact of a delay or failure of an air mission may ripple through the entire plan.

This effort built on the successful DARPA/AFRL/IF sponsored JFACC Active Technologies program. During that program it was demonstrated that Active Technologies could tie together disparate data sources to provide both planners and operators with timely, critical information on which they could base decisions. However, the DARPA JFACC architecture was completely unlike the current air operations environment in that it relied on advanced object-oriented databases and communications protocols. The challenge was to transition the capabilities demonstrated in the futuristic architecture of the JFACC program to the "real-world" environment of today's Air Force Operations.

The JFACC Active Technology program built upon a solid foundation of research by Dr. Sharma Chakravarthy. BBN Technologies and Dr. Chakravarthy have been collaborating on application of Active Database Technologies for over seven years.

2.1   Technical Challenges in Managing the Flow of Information

2.1.1 Identifying Information Needs

Current technology leaves the identification of information needs in the hands of users. It is up to the user to recognize the situation, evaluate the current context and determine the information that is critical to decision making. Users who lack training and experience are prone to errors or delays in recognizing the situation, evaluating the context and identifying the required information.

2.1.2 Information Extraction

Once the need for information is defined, the source of the data must be identified. One of the more subtle issues involved in Information Extraction is that most current systems and operators assume that all information required is contained in a single data source. In reality, few information requirements can be fully met with a single data source. When multiple data sources are identified, there is no way to resolve conflicts between the sources other than through past experience. Issues of data freshness (timeliness of updates) and accuracy cannot even be properly addressed when the same data resides in multiple data sources. In addition, most query systems erroneously assume that the operator is fully knowledgeable in the contents and structure of the data source.

2.1.3 Knowledge Integration

Current technology leaves Knowledge Integration completely to the human agent. Users go through a series of manual information extraction activities, evaluate and compare the results, and apply experience and intuition to come to a reasonable conclusion. As less experienced players are asked to play a role, errors and delays in Knowledge Integration will multiply.

2.1.4 Recognizing Change

The most critical factor in recognizing change is awareness that a change has occurred. Most of today's stovepipe systems include at least a minimal alerting capability, but none are capable of evaluating that change outside their own narrow purview. The basic approaches used in both currently fielded systems and even in advanced research systems are problematic and cannot easily address complex events across heterogeneous databases.

2.2 Event-Condition-Action Paradigm

Active databases implement a rich and powerful "Event-Condition-Action" or ECA paradigm. Oracle and Sybase triggers correspond to primitive "Events". Composite events combine two or more primitive events using a full range of logical (e.g. AND, OR, XOR, NOT) and temporal (e.g. absolute, relative) links. Conditions can be as simple as a valid numerical range or more complex e.g. executing queries to determine the current state of the world. Actions range from simple notification alerts that an event has been detected that met the specified conditions to a complex cascade of queries and actions. Complex actions might execute database queries, trigger additional ECA rules, activate intelligent agents, etc., resulting in a "complete" report to the planner of the impacts of the detected event and possible mitigating actions. We can employ active database technologies to address the fundamental challenges of information management.

3 System Architecture

The primary components developed or modified for this effort include an Oracle Interface, the ECA Server, and an Alert Interface.

3.1 Oracle Interface

The Oracle Interface was implemented as an Oracle External Procedure (XPROC) that makes use of the Oracle Listener functionality.

3.2 ECA Server

3.2.1 Introduction

The ECA Service architecture combines the technology of the Local Event Detector (LED), the grammar and syntax of the Event-Condition-Action (ECA) schema, and the power of a distributed, ORB-based, object architecture to support active monitoring of, and reaction to, events of critical interest. Building on this service architecture, we extend the client interface to provide flexible functionality for diverse domain needs. The development areas we pursue are selected to support the use of ECA Services through graphical tools and well-known mechanisms of event communication. Our Technology Integration Experiments (TIEs) within the JFACC program group demonstrate our success in achieving our design goals.

We focus on four areas which are important to our success as a user service: client support methods in the ECA Server interface, the persistence of ECA Rules beyond the Server process, architectural extensions allowing special-purpose code to be executed as a plug-in, and support for a variety of communication paths by which clients can receive information from the Server. We build client programs which test these areas and exercise the infrastructure that supports distributed notifications.

3.2.2 The CORBA Interface of the ECA Server

The PlanServer was the first client of the ECA Server. The Notify method of the ECA Server interface is called by the PlanServer to propagate trigger notifications to the ECA Server. The ECA Server receives all triggers issued by the PlanServer, and filters the triggers through the LED. Additional methods in the ECA Server interface allow client applications to control the way triggers are filtered in the LED. Through the expanded interface, clients can define primitive and compound events of interest. Events are updates or changes in database items, which may occur singly or together with other changes, possibly in a defined temporal sequence. Clients can also define condition and action functions by name.

With these components, event-condition-action, rules can be defined to associate events with conditions and actions - the ECA Rule. When an event occurs, the condition evaluation is performed by the ECA Server, and the associated action is executed by the ECA Server if the evaluation produces a specified result. Notification to the client is filtered, based on the outcome of the condition evaluation.

The ECA Server interface also provides methods to manage defined rules: to list or delete rules, to toggle rules on and off, or to reset parameters for rule execution. This expanded interface for rule creation and management provides a basis for tools which build and maintain an archive of ECA Rules. Our TIE illustrates how such a tool might be built to manage rules for a specific domain, such as weather.

3.2.3 Persistence of ECA Rules

ECA Rule objects are not persistent in the LED. The LED exists for the life of the process in which it is created, in this case the ECA Server process. When the server exits, the LED no longer exists, and all objects that were created within it are gone. A shadow persistence for ECA objects and rules was defined and built in an object-oriented database. When the interface methods of the ECA Server are used to define ECA events and rules in the LED, a corresponding set of objects is created in shadow persistence. The potentially complex relationships between events are also maintained, as well as definitions for condition and action functions. All ECA components and their relationships are reconstructed in the LED at startup of the ECA Server. A cross-reference between persistent and transient ECA objects is maintained for the life of the LED, so that any change made to the LED object is also reflected in its persistent shadow object.

Objectivity was selected as the COTS OODBMS to implement the shadow persistence of LED objects. We have worked with it on many projects and have found it to be a dependable tool in project development. Among noteworthy features are: use of handle indirection to prevent data corruption through the API; benchmark performance and scalability documented for large and intensive applications; a rich set of support features, including partitioning, replication, and schema evolution; excellent technical support; and adoption by mission-critical projects in both research and industry.

3.2.4 Plug-in Architecture for Dynamic Functions

The condition and action functions which the ECA Server executes on behalf of clients, are specified through the Server interface, and selected for association with events when ECA Rules are created. In the original Server, condition and action functions were coded directly into the ECA Server itself. The limitations of this approach are clear. The ECA Server would have to be

recompiled if any function was changed or added, and errors in the functions might cause the Server to crash. Client applications are best served if they can develop their own condition and action code, associate their code with events they define, and then ask the ECA Server to execute the code on their behalf when events occur.

Because the current version of ECA Server is written in C++, we are able to use the convenient dl functions provided with Solaris to load functions dynamically into the ECA Server process. Dynamically loadable libraries of functions can be built by any client and placed into a library directory. The client can use the Server interface to define ECA conditions and actions, which specify the name of the library and function name. The ECA Server will find the named library and load the specified function from it. A comparable plug-in scheme could be implemented for a Java version of the ECA Server, if we were to build one. It would not be quite as simple as using the dl utilities, however.

Default condition and action functions, which perform some simple evaluations and notifications, are also provided by the ECA Service. Any client may use these functions in a rule definition.

3.2.5 ECA Server Interfaces

In this section, we describe the ECA Server interfaces that are exported via CORBA to clients of the ECA Server. The ECA server interfaces are divided into four categories: Rule, Condition and Action, Event Management, and Event Detection. These interfaces are depicted in Figure 3-1.

The rule interface consists of operations that create composite events, and ECA rules. Clients, such as the Rule Editor, create new rules in the ECA server operation primarily using this interface. The Event Management interface allows clients to disable or enable individual rules, or to change rule execution contexts, and priorities. Like the rule

Figure 3-1

interface, the primary client of this interface is the rule editor client. The Condition Action interface allows conditions and actions to be loaded into the ECA server dynamically during the ECA server execution. Finally, the Event interface allows primitive events to be delivered to the ECA server.

One additional interface is the OODB access interface. However, this interface is private to the ECA server and not exported.

3.3 Distributed Alert Monitor Server/Client

The Distributed Alert Monitor is a simple Java HCI designed to provide a client with a list of recent alerts and provide the ability to view the background information related to them. JDK version 1.2.2 was used to develop the software. Designed primarily as a tool for demonstrating the concept of distributed alert notification, the program is very small, easy to use, flexible, and platform independent. The alert source for the demonstration was the ECA Server; however the Distributed Alert Monitor was designed to accept input from a variety of sources. It is the responsibility of the source program to convert the alert data into the proper format.

Java was chosen as the development language because it provided well designed toolkit for both the HCI components and the distributed delivery system. The Java RMI package was chosen as the distributed delivery system, because it was robust enough to demonstrate the concept and easy to implement. The HCI component was developed using the Java Swing package, which is the standard java HCI toolkit.

Figure 3-2

The client side is a very small program that takes input from the source system in the form of a parameter list. The parameters include:

a. Type/Area of Interest
b. Priority/Text
c. URL
d. Name
e. Destination Hostname
f. Archive
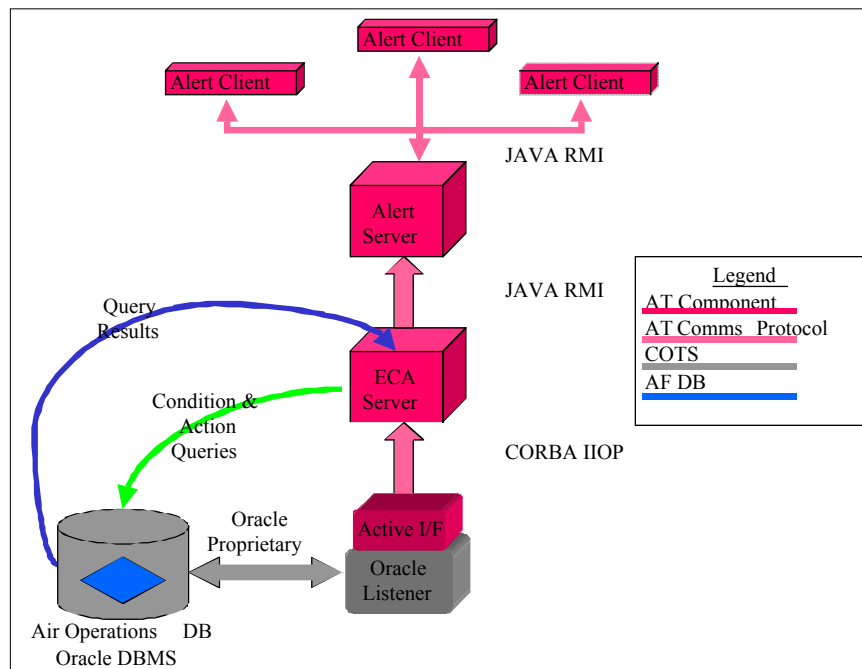
The Type/Area of Interest parameter is a concatenation of two features of the alert object, divided by a colon (:). The three types include URL, MAP, and TABLE. The Areas of Interest currently include Strike Planner, Airlift Planner, Force Support Planner, ISR Planner, OSA

7

Planner, and JFACC. The Type of alert is predefined in the Distributed Alert Monitor, however the Areas of Interest are derived from a grammar file and can be defined "on the fly."

The Priority/Text parameter is a concatenation of two features of the alert object, divided by a colon (:). The Priorities include RED, YELLOW, and GREEN. The Text parameter is the alert string displayed on the Distributed Alert Monitor HCI.

The URL parameter is the http address of the Web data associated with URL type alerts.

The Name parameter is a unique identifier for the alert.

The Destination Hostname is the DNS or IP address of the Distributed Alert Monitor Server.

The Archive is a jar file containing MAP and/or TABLE data for those alert types.

The Distributed Alert Monitor client, using the Java RMI package, assembles these parameters and pushes the alert to the server. If the server side is not running, the client program will fail.

The Distributed Alert Monitor server is a program divided into five parts:

    a. RMI Methods
    b. Area Of Interest Selection Window
    c. Alert Monitor Display Window
    d. Openmap toolkit
    e. JClass toolkit

There are also three supporting files associated with the server, a standard Java properties file, a standard java policy file, and the Area of Interest Definition file. The properties file is used to specify the Server Hostname, the Web Browser, the Area Of Interest Definition filename, and the root directory for MAP and TABLE archives. The policy file is used to control RMI access features. The Area Of Interest definition file contains those Areas Of Interest to be displayed in the Selection Window. Figure 3-3 is an example of the Java property file used by CACC to configure the users environment. The format of the file differs slightly by the computing platform, the example shown is for a Windows-based system.

```
# This is the properties file for the cacc alert monitor

# Where is the alert server running?
# can use dns or ip address
HostName = 128.132.33.131
# Which browser to view alert data, and where is it (spaces ok in
path name)
Browser = \\Netscape
# Which file is being used to retrieve Mode list and where is it
Grammar = \\Files\\caccAlertMonitor\\rmiVersion\\grammar.igr
# Where is map and table data being stored?
# Must have trailing \\
Archive = \\Files/ALERTS\\
```
Figure 3-3 CACC Java Properties Files

The RMI Methods are those methods used to move the alert information from the client to the server.

The Area of Interest Selection Window is the entry point of the server program. While this window is displayed, alerts generated by invocation of the client program are received and archived. The window displays a list gleaned from the Area Of Interest Definition file. Once the user selects his Area Of Interest, this Window is destroyed and the Alert Monitor Display Window appears. The Alert Monitor Display Window is populated with all alerts of the appropriate Area Of Interest which have been received while the Area of Interest Selection Window was active, as well as any which may subsequently arrive. The alerts are displayed in ascending order with the most recent displayed first (on top). The user can view the contents of an alert by clicking on the alert text. Clicking on an URL Alert brings up a web browser (which browser is defined in the properties file) displaying the contents of that URL. Clicking on a MAP alert invokes OpenMap, the Map (and underlying Table) data is retrieved from the archive jar file associated with the alert. Clicking on a TABLE alert invokes JClass to display the table data retrieved from the archive jar file associated with the alert. The archived data (jar files) are unjarred and saved in a subdirectory of the directory specified in the properties file.

3.3.1 Needed Enhancements

There are several weaknesses in the current design which need to be addressed if the Distributed Alert Monitor is to be deployed for more than demonstration purposes. While archived data for alerts is stored on the server file system, the alert list is not persistent. If the Server is terminated, the existing alert list is lost and cannot be recovered. There is no functionality to retrieve and display the archived data. Also, there is no programmatic method available to remove unwanted archives.

9

Another weakness is that the server side must be running in order for the client side to be able to send alerts. Obviously what is required is for the alerts to be maintained on the client side until such time as the server is available (i.e. message waiting).

There is also currently no way for the server to acknowledge receipt of alerts. It also would be useful if the server were to inform the client when an alert has actually been viewed (i.e. message receipting). Also, there is no "on-line" status awareness between the client and server.

These features were not required for the proof of concept phase, however they would have to be part of a truly functional Distributed Monitoring system.

3.3.2 Design Alternatives

The possibility of using a COTS product such as one of the Instant Messaging Programs (ICQ, Yahoo Messenger, etc) was explored. These tools provide on-line status awareness and provide for message waiting when the destination side is not active. However, using one of these tools would severely limit the possibilities of expanding functionality in the future, and would be difficult to implement without access to the source code. Another potential tool is Jabber. Jabber's main advantage is that it defines a protocol for bi-directional internet communications, which also provides for on-line status awareness, message waiting, and message receipting. The disadvantage of using Jabber lies in the fact that there are few existing servers and clients, and the ones that exist are platform specific and third party programs. Using them would involve maintaining code from several external sources, Jabber, the client, and the server. The other alternative of joining the Jabber Development Team and creating a platform independent client also exists. However, since Jabber is currently in a pre-release version, the potential for bugs in the software is very high, and the level of effort required would be quite high relative to the functionality desired. Once the Jabber program has reached a greater maturity level, it should be revisited.

Lessons Learned

The RMI Architecture works very well for a one-to-one client server relationship. However, for a many-to-one relationship, RMI probably is not the best approach. Other approaches, such as Java Enterprise Beans should be investigated.

4   Evolution of the CACC Demonstration Architecture

4.1   Past Successes building for the future

This effort was be the fruition of over ten years of research and prototyping of Active Database technologies under the auspices of Office of Naval Research (ONR), Air Force Research Lab (AFRL), Rome, NY, SPAWAR Systems Center, and DARPA. Dr.
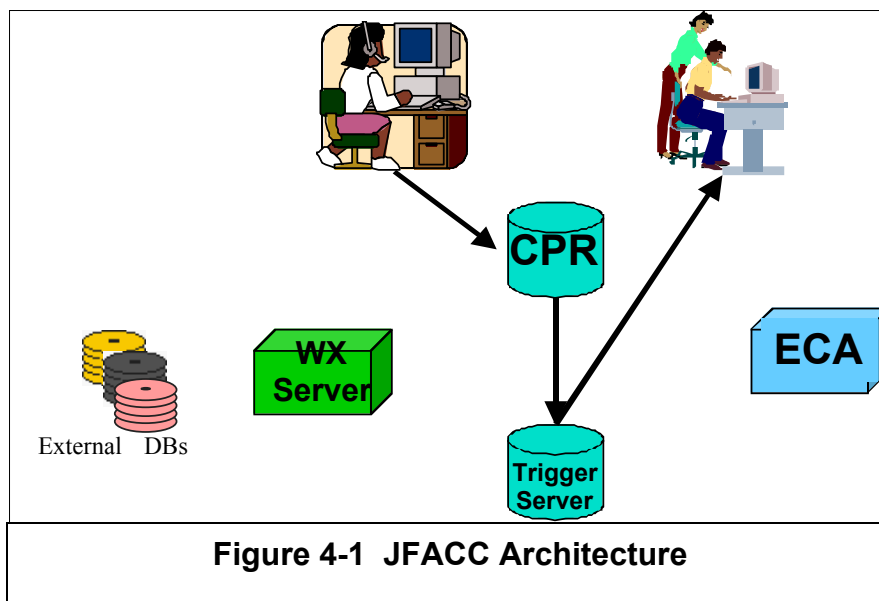
Figure 4-1 JFACC Architecture

Chakravarthy has been involved in Active Database technology research for over ten years. For the past eight years, BBN has collaborated with Dr. Chakravarthy, first to identify potential applications for Active Database technologies within the Military Command and Control domain and then to implement active technologies. This culminated in DARPA's JFACC Active Technology project that demonstrated the comprehensive situation monitoring across multiple data sources.

The JFACC program was built upon an advanced technology architecture developed under DARPA's JTF ATD project. The architecture included a Plan Server (CPR), a Trigger Server, and made use of CORBA protocols for inter-process communications. The Active Technologies effort focused on developing a Service that provided situation monitoring for a broad
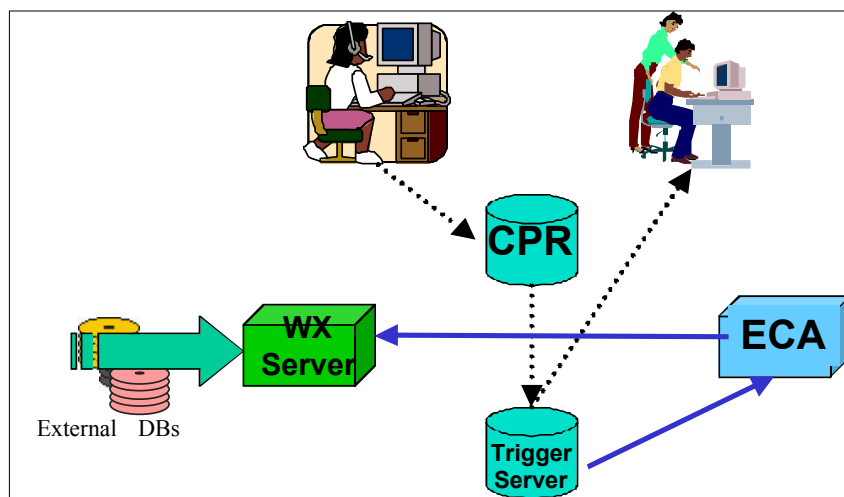
Figure 4-2 Identifying Change and Impacts

spectrum of JFACC users. Figures 4-1 through 4-3 show a simplified view of the JFACC Active Technology Architecture and Demonstration. Figure 4-1 shows how changes in the Plan Server made by one user were delivered to other interested users. The notification contained no

explanatory information on what was changed, what the impacts were, etc. Figure 4-2 shows the ECA Service identifying the change (in this case a new mission schedule) and triggering weather situation monitoring rules. The Weather Server pulls weather data from external databases and evaluates the weather monitoring rules (the rules were specific to mission type).

Figure 4-3 shows the delivery of changed information and the impact of weather (if appropriate) to interested users. In addition, the ECA Server continued to monitor the mission from the time it was scheduled until it was completed as weather forecasts were updated.

Figure 4-3 Delivery of Knowledge

The primary focus of the JFACC Active Technology experiments was in relating external information in the form of real-world weather data to both Course of Action development and Plan execution. This was successfully demonstrated during the JFACC Phase II demonstration. The key capability demonstrated was the transformation from a process of delivering disparate pieces of data to users that they must then correlate to

Figure4-4 CACC Active Technology Architecture

delivering the information and knowledge necessary to begin the decision making process.

The CACC Active Technology Demonstration took this one step further when it transitioned the JFACC prototype to the Theater Battle Management Core System architecture. The ECA Service was integrated with the Air Operations Database (Figure 4-4). The key capabilities demonstrated included prototyping interfaces to external applications and the range of alert delivery

mechanisms that were demonstrated (e.g. geographic, tabular, URL, email, pager). The interface to ICIS provided sustainment analysis of Air Battle Plans. The status of each component is indicated by the color of the connecting lines in Figure 4-4 (**Green** – prototype implemented, **Yellow** static demonstration, **Red** – not implemented). Over twelve rule categories were demonstrated and over twenty weather-air operations rules were implemented.

5   Process and approach to CACC scripts

The 'AODB Based ECA Rules' document is a living document that defines the rules for this project. The document also defines primitive events as database table inserts or updates to the Air Operations Database.

All rules are defined by activities in the Air Operations Database (stored in Oracle).

Approach:

ECA rule conditions, actions, and simulated events were implemented as scripts for both testing and demo purposes. These scripts were implemented as Unix shell scripts and Structured Query Language (SQL) scripts, as appropriate. We chose not to store these scripts in Oracle, since we anticipated that the system will grow as non-database events, conditions and actions are added to the system. In addition, we allowed the possibility of supporting databases other than Oracle. By co-locating scripts, we keep control and rule-related data in one place.

Events:

Design – Database triggers return the *rowid* of the record that was inserted or updated. (This design should be changed for long term use. The primary key[s] should be returned). This rowid approach was chosen for this initial run so a generic trigger could be implemented across all tables that needed triggers.

Implementation – Triggers are stored in the database.

Conditions:

Design – All condition scripts, SQL included, are stored in a directory. SQL scripts return T or F, depending on whether or not the condition is met. There are two scripts for each rule condition, one that controls the process (script.cXX) and the other, which performs the process (ucX.sql).

Implementation – The condition is tested via scripts in ~jkraska/cacc/src/conditions. Most of the conditions are called in this order: script.cXX, which calls runX, which then calls ucX.sql; where X refers to the AODB-ECA case number. Please note that this is an area where scripts could be improved.

Actions & DB-Actions:

Actions – Design – They set up the environment, creating needed directories, etc., possibly calling for a database action (defined below).

Actions – Implementation – Joe wrote the action scripts.

DB-Actions – Design – This section follows the separate control – process approach introduced earlier. The controller scripts set up the environment and call the SQL scripts in the proper order. The processing of the database actions may require multiple SQL scripts so they are stored in a directory, the directory naming convention: sql_X.

DB-Actions – Implementation – For each rule that is supported there is a set_X.sql file (controller) and a sql_X directory (repository for specific SQL scripts). The set_X.sql file sets up the environment and then calls the SQL scripts in the sql_X directory in the proper order.

Testing scripts: As an aid to testing and giving demos, there are a number of SQL scripts that simulate the types of actions that should trip an event.

Completed Work

The document AODB-ECA Cases defines 16 ECA Cases. The table below indicates the level of

Table 1: Completed Work

completeness for the individual rules.

#	testing script	event	cond.	action	db-action	AODB Based ECA Rules
1	X		X^	X	X	Time Critical Target: Urgent Ground Attack Request
2	X		X^	X	X	Time Critical Target: Urgent JTASR Request
3	X		X+	X	X	Weather Monitoring: New Mission
4			X-	X		Planning: Sustainment Monitoring
5			X-	X		Planning: Climatology
6				X		Execution: Sustainment Monitoring
7	X		X^	X	X	Mission Status Change

8	X		X^	?		Mission Delay (Estimate)
9	X		X^	?		Mission Delay (Actual)
10				?		Weather Monitoring: Mission Update
11			X+	X	X	Weather Monitoring: Mission Event Completion
12			X+	X	X	Weather Monitoring Mission Status Change
13	X		X^	X	X	Weather Monitoring: Weather Update
14				X		Weather Monitoring: Weather Insert
15			X^	X	X	NOTAM Monitoring: Event Insert
16				X		NOTAM Monitoring: Continuous Monitoring
17				?		?? Looks like a version of rule 7 ??
						Db-action - These scripts can be found in ~jkraska/cacc/src/dbactions. For each rule that is supported there is a set_X.sql file (controller) and a sql_X directory (repository for specific SQL scripts). The set_X.sql file sets up the environment and then calls the SQL scripts in the sql_X directory in the proper order.
						Action - These scripts can be found in ~jkraska/cacc/src/actions. There are scripts for all rules, there is no indication whether the script is active or just dummied up (the ? indicates that it seems dummied up).

		Cond.: These scripts can be found in ~jkraska/cacc/src/conditions. An X^ indicates a calling sequence of script.cXX -> runX -> ucX.sql. An X+ indicates that the runX command (a sqlplus call) has been incorporated into the script.cXX file. An X- indicates that the default script.cXX (always returns T) is being run. The SQL scripts are available, in this directory, for that rule. However, the db-action scripts have not been created.
		Event: These are represented as triggers that are stored in Oracle.
	Testing script: These scripts can be found in ~jkraska/cacc/src/triggers. An X indicates there is at least one script that is used for testing/demo.	
#: This corresponds to the 'AODB Based ECA Rules' document numbering of the rules.		

Future work

Relocate scripts and code under a more appropriate directory structure.

The database triggers need to be rewritten to return the primary key[s]. The scripts that rely on this information also need to be rewritten (this should be isolated to rewriting the scripts that create the defines used in SQL scripts).

Some of the condition scripts need to be condensed.

The remaining scripts and code needs to be written.

Appendix A: Active Database Infrastructure

Event-Condition-Action Infrastructure

Paul C. Neves

1 Introduction

The purpose of this document is to present an overview of the JFACC Event-Condition-Action (ECA) server infrastructure. We will highlight some of the design decisions and issues surrounding our infrastructure design and implementation. We also present our lessons learned from our design and implementation. Finally, we present some possible future directions based on our lesson learned.

In section 2, we present the method we used to select a preliminary approach to the design of the ECA server infrastructure. In section 3, we present the issues and design decisions the design of the infrastructure supporting the integration of the University of Florida's Event-Condition-Action evaluator. Next, in section 4, we examine the changes necessary to integrate a JFACC server (in particular the Plan Server) into the ECA infrastructure. We also present the types of ECA events and these ECA events are communicated between the ECA Server, the Plan Server, and possibly the clients. In section 5, we discuss the multi-threaded implementation of the ECA server. Finally, in section 6, we present lessons learned and future directions.

2 Preliminary Approach

In this section, we specify the problem statement for the ECA server. We present two approaches to an ECA server design to solve the problem statement above. Included are the design criteria and experiments that led to our initial approach.

2.1 Problem Statement

Our problem is to integrate Event-Condition-Action technologies into the JFACC system. This means that we must combine the following components:

Distributed Computation Framework – The JFACC system is based on the CORBA distributed object framework standard.

JTF ATD Reference Architecture – JFACC is using components of the Joint Task Force's Advanced Technology Demonstration architecture for planning and plan execution. In particular, we are using the Plan Server and Stream Trigger Server.

Sentinel Project's Event-Condition-Action Evaluator – University of Florida's ECA evaluation technology that supports their active database management research.

Persistent Rule store – Objectivity object-oriented database management system used as the rule persistence for the University of Florida's ECA Evaluator.

The combination of these components must be done in such a way that we, first and foremost, maximize the utility of the ECA service to the JFACC program. That is, the design of the ECA server should add value to the JFACC program by consolidating planning and plan execution integrity constraints that simplify the development of applications and other JFACC servers. Second, our design should maximize the performance of the system by using resources efficiently. Next, our design should not be built to support a particular application, but also a wide range of CORBA applications – both JFACC and non-JFACC. Finally, our design should make the ECA server easy to use and minimize the time to develop the server.

2.2   Alternative Architectures Overview

This section gives an overview of the to alternative architectures under consideration. Each alternative has the following components: client, Plan Server, ECA Server, and Socket Trigger Server.

2.2.1 Combined Planserver/ECA Server

The Testbed architecture of the combined Planserver/ECA server is shown in Figure A-1. The motivation for this architecture is to locate the ECA rule processing close to the objects for which events are being monitored. In addition to the current interfaces implemented by the Plan Server, a skeleton ECA interface is added to the Plan Server. The ECA interface and implementation is responsible for the following:

1.   Implement a method to set the average compute time to evaluate an ECA rule.
2.   Implement a method to set the probability that an ECA rule fires.
3.   Simulate the firing and evaluation of an ECA rule.

A typical client/server interaction starts with the client invoking an "update" request on the Plan Server which in turn updates the object, and send a trigger notification via the Socket Trigger Server to the invoking client. Next, the ECA portion of the Plan Server determines whether a rule should fire according to the ECA rule firing probability, and, if necessary, fires the rule and spins for duration equal to the average compute time for an ECA rule. A trigger notification is sent to the invoking client after an ECA rule is evaluated.

In the Testbed, the clients are responsible for setting the average compute time to evaluate an ECA rule, and the probability that an ECA rule fires prior to an "update" invoke. Each Testbed experiment below describes how these values are determined and when they are set.

2.2.2 Separate ECA Server

The Testbed architecture of the separate ECA server is shown in Figure 2. The motivation for this architecture is to locate the ECA rule in a separate server so that updates and ECA rule evaluation can proceed in parallel on two different processors. The separate ECA server

Figure 2 Separate ECA Server Testbed

implements the same skeleton ECA interface described for the combined ECA server.

As in the combined Plan Server/ECA Server architecture, a typical client/server interaction starts with the client invoking an "update" request on the Plan Server which in turn updates the object, and sends a trigger notification via the Socket Trigger Server to the invoking client. However, in the Separate ECA Server, a trigger notification is also sent to the ECA Server. The ECA server determines the triggering object and whether a rule should fire according to the ECA rule firing probability, and, if necessary, fires the rule. The ECA server obtains the old and new version of the triggering object and spins for duration equal to the average compute time for an ECA rule. Finally, a trigger notification is sent to the invoking client after an ECA rule is evaluated.

In the Testbed, the clients are responsible for setting the average compute time to evaluate an ECA rule, and the probability that an ECA rule fires prior to an "update" invoke. Each Testbed experiment below describes how these values are determined and when they are set.

2.3   Evaluation Testbed Environment

2.3.1 Hardware and Software Platforms

One UltraSPARC 167 MHz running Solaris 2.5 with 128 Mbytes (kali.sd.bbn.com)

One UltraSPARC 167 MHz running Solaris 2.5.1 with 128 Mbytes (hal.sd.bbn.com)[2]

SunSoft C++ Compiler version SC4.0 18 Oct 1995 version 4.1

TCP/IP Throughput test TTCP (BRL)

Planserver -- Modified E5 version to support ECA interface

Socket Trigger Server – E5 version

Experimental Separate ECA Server Version 0.1

10 Mbs Ethernet -- to T20 gateway -- to 10 Mbs Ethernet (lightly loaded)

2.4   Testbed Environment Assumptions

All servers and clients are single-threaded.

> *Rationale: Many lines of code would have to be made thread-safe to explore multi-threaded clients and servers. However, some of the single-threaded results can later be applied to the multi-threaded scenario, or used to setup a multi-threaded simulation.*

The focus of the evaluation is on the performance of a single object update.

> *Rationale: This assumption limits the scope of the evaluation, and allows applications to estimate their performance bounds based on object update patterns, if necessary.*

Webserver garbage collection of orphaned object versions is disabled throughout the tests.

> *Rationale: Disabling garbage collection eliminates collection effects on the results, although none of the performance tests cause the garbage collector to run.*

Socket Trigger Server and its associated clients are co-located.

> *Rationale: Socket trigger servers can either reside on the same host as its clients or a remote host. A co-located socket trigger server and client results in less LAN message traffic and better performance.*

---

[2] For some reason, a loop that executes in 1 millisecond on the machine kali executes in 2 milliseconds on the machine hal. These machines are supposed to be identical.

2.4.1 FIFO Performance Experiment

UNIX FIFOs that is, named pipes are delivery channels for sending trigger notifications between a Socket Trigger Server and a co-located registered client. The FIFO performance measurements give a lower bound on the delivery throughput between the server and its registered client.

We measure a FIFOs throughput by dividing by N the elapsed time required to send N packets containing M bytes of data to a receiver.

2.4.2 TCP/IP Performance Experiment

We use the TCP/IP performance tests to measure the throughput of an UltraSPARC TCP/IP implementation and the TCP/IP data throughput between the two machines. We did not measure UDP performance, since the protocol is not used by any of the components in the Testbed.

We measured the TCP/IP throughput using the TTCP benchmarking tool written at the US Army Ballistics Research Lab (BRL). Instances of the TTCP program act as a data sink and also as a source to transfer data consisting of fixed-sized packets determined by command-line arguments. Memory caching effects are not taken into account during these tests.

The results of the TTCP provide performance numbers on the lowest layer of the Testbed, and are applicable to all components.

We measure the TCP/IP throughput in kilobytes per second for both intra-host and inter-host connections.

2.4.3 Null Method Invocation Performance Experiment

The Null Method experiment determines the lower-bound cost associated with a cross address-space Orbix method invocation. We measured the cost of two types of method invocation: one-way and two-way invocations. Two-way method invocations are the most frequently used in the Testbed. One-way invocations deliver events to a Socket Trigger Server.

In Null Method experiment, we measure the elapsed time to perform 100 null method invocations on a server, and compute the mean response time for each invocation. We repeated this experiment 10 times for both one-way and two-way method invocations.

We measure the mean response time for both one-way and two-way null method invocations.

2.4.4 Socket Trigger Server Performance Experiment

The Socket Trigger Server experiment determines a lower-bound response time from event generation to event delivery via the Socket Trigger Server. We considered two cases: the co-

located case, and the remote case. However, since the results in both cases were so similar, we report only the results of the co-located case.

The experiment involves the Socket Trigger Server and a single client. The client creates a trigger object and uses the X event loop to receive asynchronous notifications over the channel associated with the trigger object. Starting from when the client sends a burst of 100 synchronous trigger notification invocations (TS_Trigger_Fire), we measure the elapsed time to send the burst, and receive all 100 notifications.[3] There were a total of 10 bursts in this experiment.

We calculated the mean response-time for the delivery of the trigger notification.

2.4.5 Combined Planserver/ECA Server Performance Experiment

This experiment focuses on measuring two items in the combined Planserver/ECA Server architecture:

The round-trip response time for ECA notification.

The response time of a Planserver "update" invocation.

The round-trip response time (1) is one component of a measure to determine which ECA Server architecture is better. All things being equal, a faster the round-trip response time implies a better architecture. The Planserver "update" invocation is another component of measure for ranking ECA Server architectures. All things being equal, an architecture that does not affect the Planserver "update" response time is better.

In this experiment, a round trip starts in a client just before an "update" invocation occurs, and ends when the same client receives the corresponding ECA notification. We measure the mean round-trip response time by invoking a burst of 100 updates on the Planserver and measuring the elapsed time to receive the last ECA notification. Next, we divided the elapsed time by100. The experiment averages the results of 10 bursts.

We obtain the "update" response time by measuring the elapsed time to perform a burst of 100 updates on the Planserver, and divide by 100. The experiment averages the results of 10 bursts.

2.4.6 Separate ECA Server Performance Experiment

The Separate ECA Server experiment and objectives are the same as those given in the section Combined Planserver/ECA Server above.

---

[3] The asynchronous trigger notification method (TS_Trigger_Fire_Oneway) was later measured and found to be twice as slow as the synchronous version. This result was unexpected and the reasons behind it are currently unknown.

2.5 Performance Results

2.5.1 FIFO Performance

This section provides comparative results of the Inter Process Communication(IPC) performance on the UltraSPARC host kali. We were particularly interested in the FIFO performance, since FIFOs are the underlying mechanism for trigger delivery between a local socket trigger server and its client.
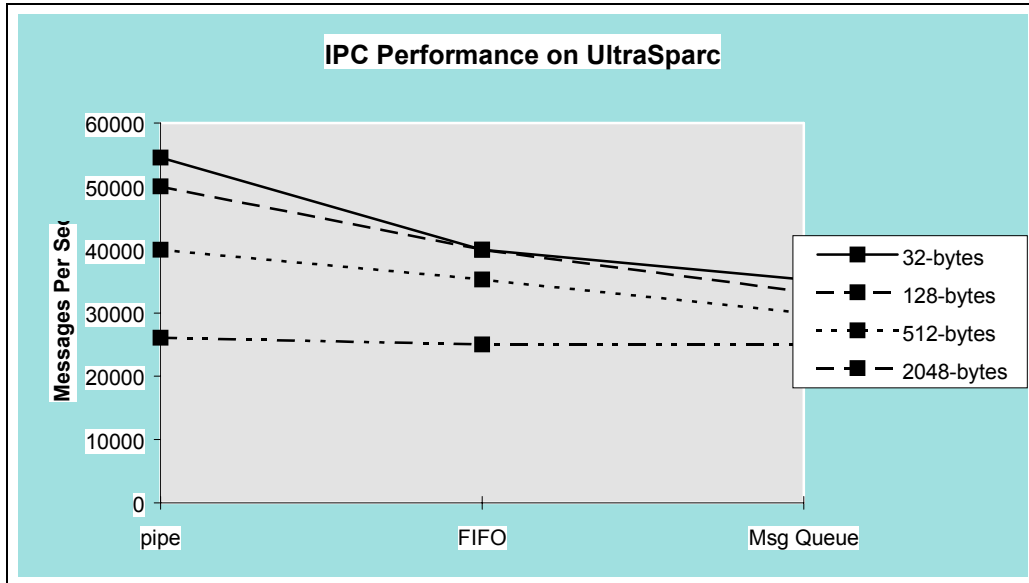
Figure 3 IPC Performance (FIFO)
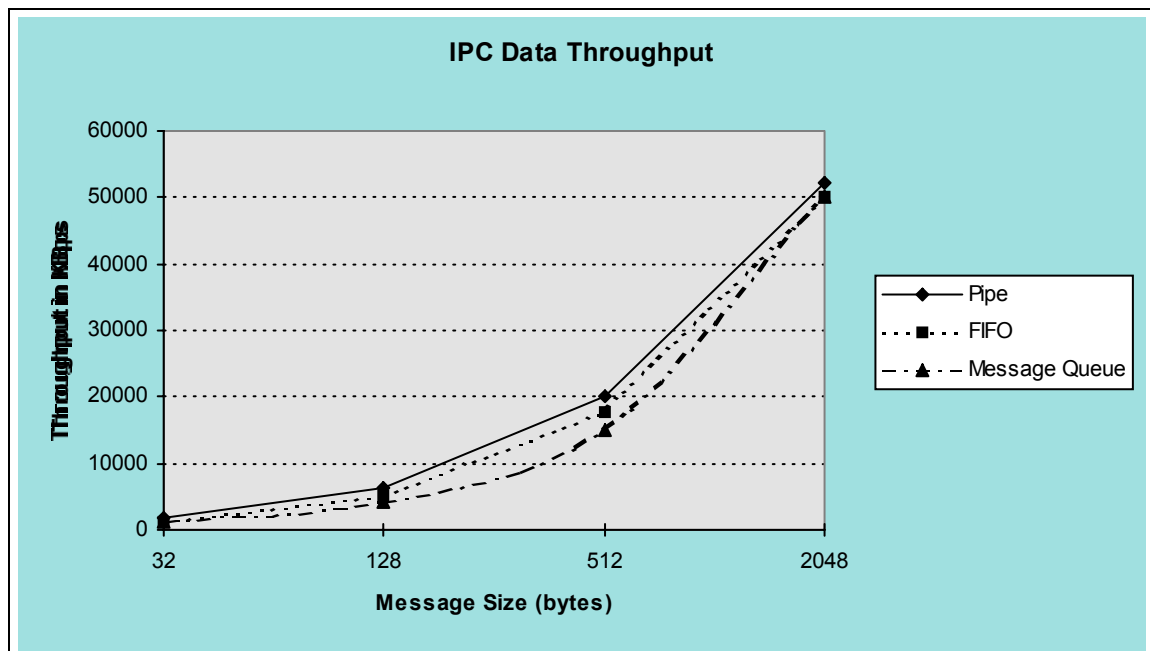
Figure 4 IPC Data Throughput (FIFO)

2.5.2 TCP/IP Experiment Results
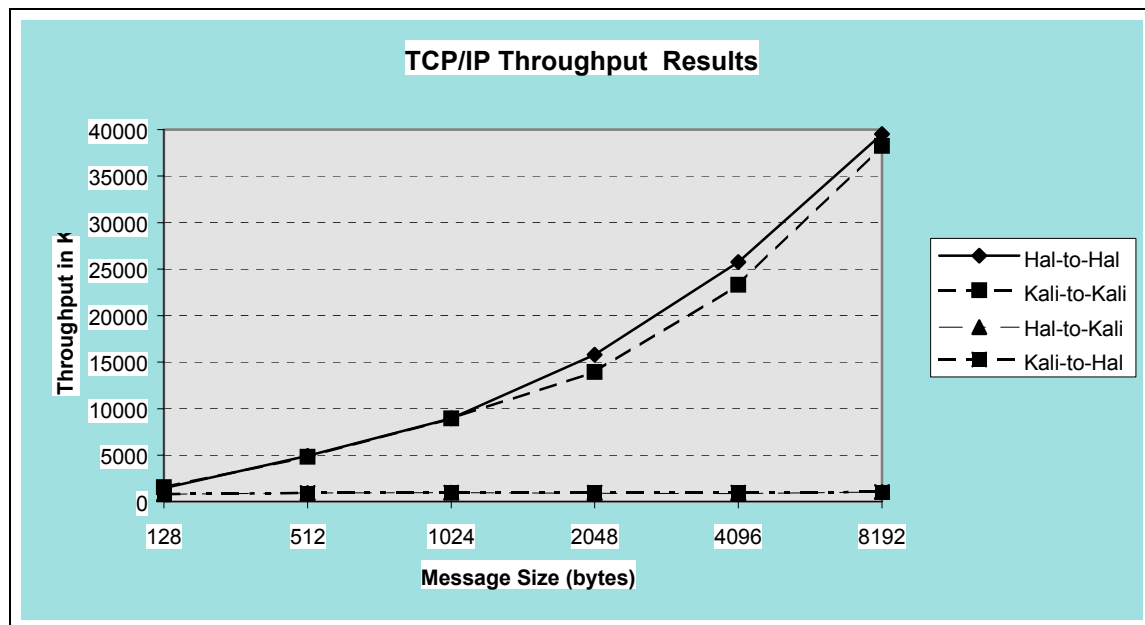
Figure 5 TCP/IP Throughput Results

2.5.3 Null Method Invocation

Invocation Mode	Mean (ms)	Std. Deviation
One-way	1.21	0.06
Two-way	1.68	0.02

Table 2 Hal to Hal

Invocation Mode	Mean (ms)	Std. Deviation
One-way	0.71	0.20
Two-way	1.23	0.60

Table 3 Kali to Kali

Invocation Mode	Mean (ms)	Std. Deviation
One-way	0.88	0.32
Two-way	1.32	0.39

Table 4 Hal to Kali

Invocation Mode	Mean (ms)	Std. Deviation
One-way	0.85	0.01
Two-way	1.70	0.16

Table 5 Kali to Hal

2.5.4 Socket Trigger Server Experiment

The socket trigger server is responsible for delivering ordered event "messages" from one client in the system to another. A client sends an event to the socket trigger server by invoking the TS_Fire_Oneway method or TS_Sync_Object_Changed (deprecated) method on the Socket_Trigger object representing an event delivery channel. The event is queued in the socket trigger server, and then delivered to the target client via UNIX FIFO.

A Socket Trigger server functional requirement is that events should be delivered in the same order in which the events were received by the server with respect to a single sender. To meet this requirement, events are queued as they are received, and sent via a UNIX FIFO to the target client. Processing of incoming events is given priority over outgoing events to avoid dropping any events. However, under heavy network and server loading conditions it is possible to drop events. The fact that incoming event processing is given priority over the outgoing event processing, and that this implementation is single-thread means that there is an unbounded delay between the time when an event enters the Socket Trigger Server, and when it leaves.

The purpose of this experiment is two fold. First, this experiment should measure the average time it takes the Socket Trigger server to process an event from entry to exit. Second, it should measure the average event delivery throughput for the local and remote event generation and delivery.

The socket trigger server requires on average 3.26 milliseconds to un-marshal a TS_Trigger_Fire method invocation, en-queue the trigger to be delivered, de-queue the trigger.

2.5.5 Combined Planserver/ECA Server Performance
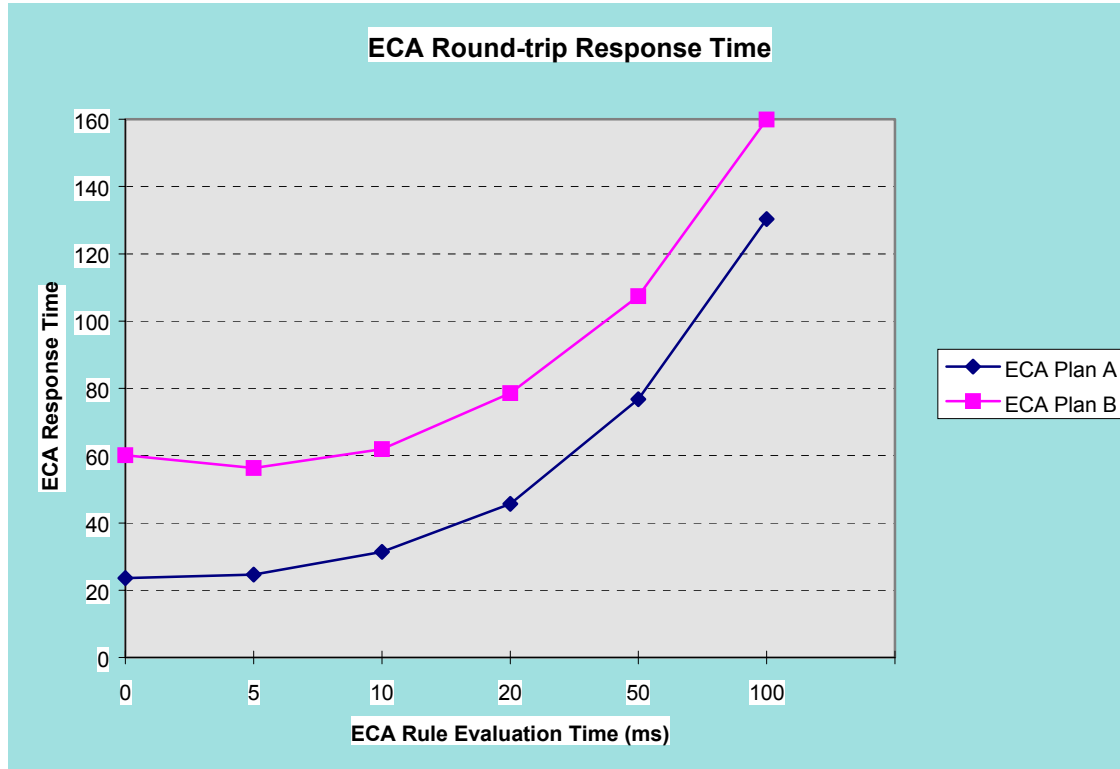
Figure 6 ECA Round-trip Response Time

2.5.6 Separate ECA Server Performance
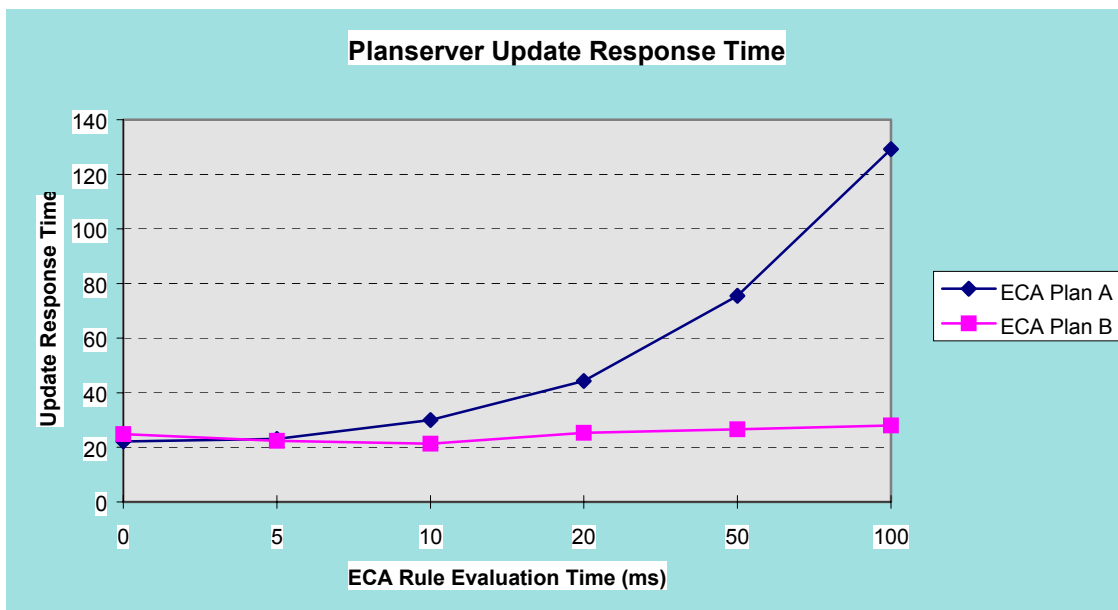
Figure 7 Planserver Update Response time

27

3   Event-Condition-Action Technology Integration

At this point, the ECA server is a skeleton prototype server with a simple built-in condition, a built-in action, and a minimal interface. The interface consists of an operation to set the probability that a rule fires, an operation to set the duration of an action, and an operation to deliver a primitive event. This section describes the integration of Event-Condition-Action (ECA) technology into the server architecture explored and selected in Section 2.

3.1   Sentinel Architecture

This section describes the technology base that was integrated into the ECA Server. The ECA technology is based on the Sentinel Architecture for Active Object-Oriented Database Management Systems (AOODBMS) developed at the University of Florida. The Sentinel project is based on Texas Instrument's Open Object-Oriented Database (OODB), the SNOOP event specification language preprocessor, and local and global event detector components. The Sentinel architecture is depicted in Figure 8.

The Sentinel architecture is a centralized architecture where all ECA code lives in the address space of the application accessing the database. The application code is instrumented with SNOOP preprocessor directives for defining primitive events, composite events, conditions and rules. The SNOOP preprocessor processes the application code, and inserts the appropriate code that implements the preprocessor directives. When the application executes, primitive events are generated and processed by the local event detector, which in turn

Figure 8 Sentinel Architecture

schedules rules to be evaluated in the context of the object database.

The global event detector is used for applications that are executing in a distributed environment.

3.2   ECA Technology Integration Approach

In this section, we describe our approach to integrating Sentinel ECA Technology into the JFACC ECA Server in a CORBA environment under the design goals specified in Section 2.1. We considered three approaches to adopting the Sentinel ECA technology: (1) we could adopt the ECA technology wholesale with essential no changes, (2) we extract core functionality and augment where necessary, or (3) we could start over from scratch. Our approach was to extract core functionality and augment where necessary. The focus of our integration effort was on the local event detector (LED) of the sentinel architecture.

Our approach was driven by the following reasons. First, the LED component represented significant debugged work for detecting primitive events, creating composite event trees, and for evaluating conditions and actions in a C++ environment. Redesigning this work would have taken significant resources. Second, the SNOOP preprocessor and event specification language was tightly coupled with a centralized computing architecture, and would not work in a multi-language distributed computing framework such as CORBA. Instead, we implemented the equivalent SNOOP directives in an interface to the ECA server[4]. Third, since we wanted to support multiple distributed data sources, we did not adopt the underlying sentinel database Open OODB. Instead, we focused on the Plan Server as the underlying database, but generalized the ECA database coupling to support any data source. Fourth, we eliminated the lock table in the Sentinel architecture, since we would explore locking mechanism that work in a distributed environment, and were not database specific. Finally, we deferred the implementation and adoption of rule scheduling, since it was not immediately necessary to meet initial JFACC goals.

---

[4] This interface is described in a companion document.

3.3   JFACC ECA Service Architecture

The JFACC ECA Service consists of ECA Server, the Plan Server, an event notification service, and the CORBA name service. Figure 9 depicts the relationship between the ECA Server and the Plan Server.

A client invokes queries and updates on the Plan Server and processes the results. A consequence of a client's invocation is an event that is sent to the ECA Server. The ECA server processes the event, which may fire a rule whose condition is evaluated, and performs some action if the rule's condition was satisfied.

The ECA server has a special client called a rule editor. The rule editor is used to create new composite events, and rules from the conditions and actions available in the ECA Server.
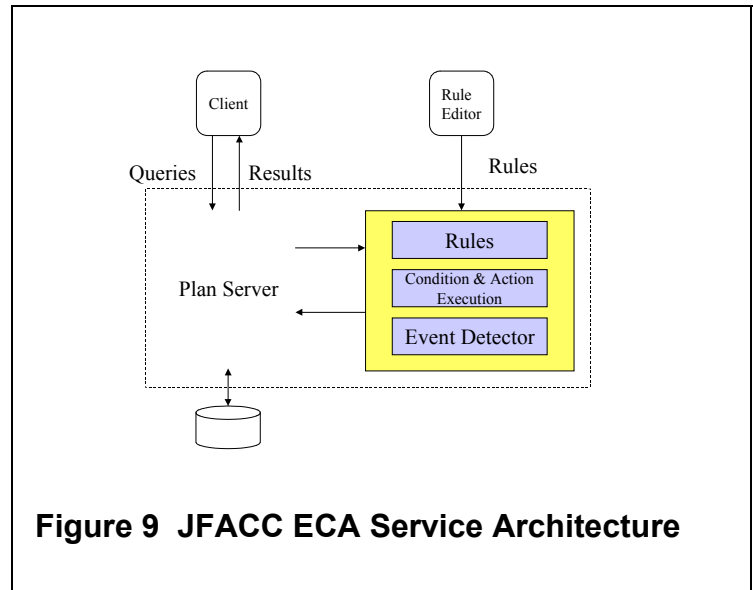
Figure 9 JFACC ECA Service Architecture

3.4   ECA Server Interfaces

In this section, we describe the ECA Server interfaces that are exported via CORBA to clients of the ECA Server. The ECA server interfaces are divided into four categories: Rule, Condition and Action, Event Management, and Event Detection. These interfaces are depicted in Figure 10.

The rule interface consists of operations that create composite events, and ECA rules. Clients,

such as the Rule Editor, create new rules in the ECA server operation primarily using this interface. The Event Management interface allows clients to disable or enable individual rules, or to change rule execution contexts, and priorities. Like the rule interface, the primary client of this interface is the rule editor client. The Condition Action interface allows conditions and actions to be loaded into the ECA server dynamically during the ECA server execution. Finally, the Event interface allows primitive events to be delivered to the ECA server.

One additional interface is the OODB access interface. However, this interface is private to the ECA server and not exported.
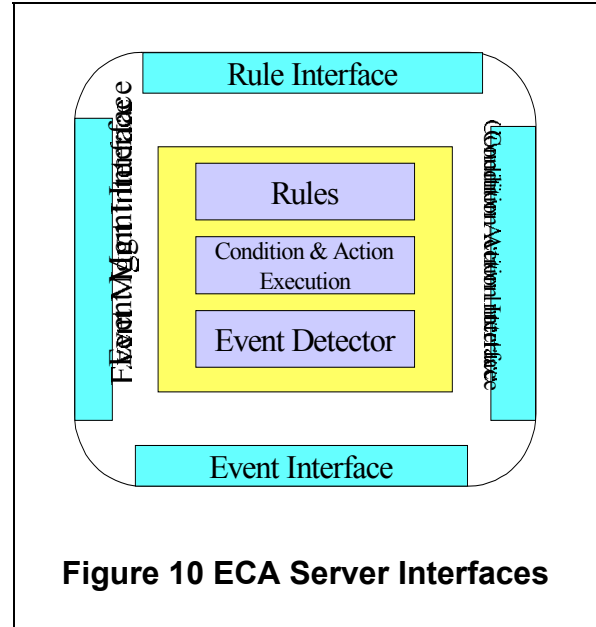
Figure 10 ECA Server Interfaces

4 Plan Server Integration

This section describes the issues associated with using the Plan Server as a source of events for the ECA Server. In particular, we are interested in the types of events generated by the plan server, the method in which these events are conveyed to ECA Server, and how the ECA server evaluates conditions and actions associated with the Plan Server.

4.1 Types of Plan Server Events

The Plan Server returns five different events corresponding to operations on objects managed by it. The events are listed in Table 5 – Plan Server Events.

Event Type	Description
Object Creation	Event generated when an object is created and initialized.
Object Deletion	Event generated when an object is destroyed. There is no event associated with the recovery of a destroyed object's resources.

Object Update	Event generated when an object is modified (cf. Creation and Deletion).
Object Version Conflict	Event generated when the Plan Server detects that an object's update has violated the concurrency control protocol.
Object Read	Event generated when an object is read.

<div align="center">

Table 5 -- Plan Server Events

</div>

In practice, the E3.6 version of the Plan Server can only generate two kinds of events: Object Update and Object Version Conflict Events. The other events are not generated because of the current implementation techniques used in the Plan Server. For example, Object Create events are not generated because the create operation is implemented using the update operation. And, there is no way to register for objects that don't yet exist. Therefore, all Object create events appear as Object update events. Ad hoc methods have been devised for clients to infer object create events from object update events, but these methods do not work for the general case.

4.2   Transmitting Plan Server Events to the ECA

Modifications to the Plan Server were necessary to convey events to the ECA Server. This section describes the changes and their implementation.

The same mechanism that the Plan Server uses to inform clients of events they are interested in was used as the basis of the initial integration with the ECA Service. Clients must register interest in particular events on particular objects with the Plan Server. This information forms the tuple (Stream Trigger Object, Plan Server Object, Event List) which is stored in a mapping in the Plan Server. After each operation, when a match is found in the mapping an event messages is fired on the stream trigger object and the client is notified.

The ECA Server uses the same mechanisms as other Plan Server clients. However, in our implementation, the ECA Server is treated as a special client of the plan server for two reasons:

ECA Server is automatically registered for all Plan Server objects.

ECA Server is automatically registered for all Plan Server object events.

All events – in the initial implementation – were delivered via the stream trigger service. Sections 2 and 3 discuss the implementation using the CORBA event services and the differences between the two implementations.

4.3   Condition and Action Evaluation

There are two approaches to evaluating conditions and actions relative to the Plan Server: remote and local evaluation. There are tradeoffs associated with both forms of evaluation. However, the ECA Service architecture supports both approaches. In this section, we discuss the remote and local approaches to condition and action evaluation, and the tradeoffs associated with each.

Remote condition or action evaluation occurs when an ECA rule's condition or action is evaluated in a server other than the ECA server (e.g. the METOC server). Remote evaluation has several advantages. First, it has performance advantages since a remote server may perform domain specific services very efficiently by design. Therefore, time consuming operations performed in a specialized remote server may be faster than the ECA server performing a local evaluation including the communication costs. Second, remote evaluation supports increased component modularity and decreased coupling because of the encapsulation enforced by process boundaries.

There are also disadvantages associated with remote evaluation. First, performance can be a disadvantage in remote evaluation if the communication costs are higher than the cost of performing the evaluation locally. Since performance can be both an advantage and disadvantage, we have left the decision to the writer of rules. Second, since the initial implementation of the ECA service was single-threaded, remote evaluation is not practical. We found that remote actions were taking so long that they were delaying the processing of queued conditions and actions.

Local evaluation occurs when an ECA rule's condition or action is entirely or primarily evaluated in the context of the ECA server. The advantages associated with local evaluation are:

1. **Uniform Representation** – conditions and actions can be specified in a uniform representation (i.e. an event specification language). This simplifies the creation of ECA service rules.

2. **Performance** – local evaluation of ECA conditions and actions eliminates communication costs. However, not all conditions and actions can be locally evaluated.

There are also disadvantages associated with local evaluation:

3. **Inflexible** – local evaluation constrains evaluation to a particular set of conditions as opposed to invoking operations on specialized servers.

4. **Implementation costs** – the ECA service will contain large portions of domain specific code thus increasing the complexity of the ECA server implementation.

4.4 Plan Server Integration Issues

We encountered a number of issues while integrating the Plan Server to operate with the ECA Service. These issues are enumerated in Table 6 – Plan Server Integration Issues.

Issue	Description
No source event filtering	The Plan Server sends all events for all objects to the ECA server. The ECA server should only receive events in which required by its rule set.
No create or delete events generated.	The Plan Server does not generate create or delete events on objects.
No condition evaluation support	Plan Server does not support ECA remote evaluation of conditions (see no query support).
No query support	One way to support condition evaluation in the Plan Server is to provide query support such as an ODL or SQL interface.
No Transaction support	The Plan Server does not support transactions. Therefore it is not possible to rollback actions performed on the Plan Server.
Undefined ECA and Plan Server startup interaction	The semantics between the ECA server and the Plan Server have not been closely examined to ensure proper operation, if a failure occurs during server startup.

Table 6 – Plan Server Integration Issues

5 ECA Multi-threaded Implementation

Although our preliminary work indicated that the performance of the single-threaded ECA server is directly linked to the performance of action evaluation, the typical action execution time was unknown. It has become evident in the Technology Integration Experiments (TIE's) that a single-

threaded ECA server adversely affects performance of ECA-based systems. This section describes our approach to implementing a multi-threaded ECA server to solve single-threaded server performance problems.

5.1 Multi-threaded ECA Server Design Goals

Our design of the multi-threaded ECA server is based on a set of key design goals. Some of these goals were imposed by the JFACC projects and other from our view of where the ECA server will be in the future. Table 7 lists these goals.

Goal	Description
Operating System Portability	Implementation must be portable to the Unix and Windows NT Operating Systems. These two systems are targets of the DII/COE.
CORBA ORB Portability	Implementation must be portable to both Orbix and Visibroker (DII/COE distributed computation frameworks).
Powerful Thread Abstractions	Thread-Model must support various server activation models, concurrency models, and portability goals. Supports experimentation with several models.
Powerful Synchronization Primitives	Synchronization primitives should be based on object-oriented design patterns and primitives found in computer science literature. For example, the primitives should allow you to build monitors, if that is the desired synchronization model.
Understandable implementation	An understandable implementation is necessary for both debugging the server, and to simplify server maintenance.
Minimal implementation effort	Supports early experimentation with different multi-threaded architectures with a fast turn around time.

Table 7 - Multi-threaded ECA Server Design Goals

5.2   Our Solution

To meet our design goals, we used freely available Adaptive Communication Environment (ACE) Framework developed by the University of Washington – St. Louis[5]. The ACE Framework provides design patterns for building adaptive object-oriented distributed communications. ACE has been used as the basis of several large telecommunication projects; it has ported to many operating systems and platforms, and has been used with both Orbix and Visibroker CORBA ORBs. Thus, using ACE as our foundation meets many of the design goals specified above. In this section, we briefly describe how we used ACE to develop a multi-threaded ECA server.

5.2.1 ECA Server Threading Model

Threads in the ECA Server can be classified into three categories: method, timer, and primitive event detector threads. Method threads are created when a client invokes an operation on the ECA server. The ECA server allocates a new thread and the client's request is passed off to the new thread. In the current implementation, a new thread is created for each new request as opposed to allocating a thread from a thread pool of previous instantiated threads[6]. There is one Timer thread in the ECA server. It is responsible for processing ECA server temporal events and manages the temporal event queue. Finally, there is one Event Decoder thread that is responsible for processing primitive ECA events generated external to the ECA server. Event decoder thread is similar to Method thread except that their synchronization points are disjoint from the Method threads. Therefore, the Event detector thread can execute concurrently with the method threads. Although both Orbix and Visibroker ORBs support this model, they use non-portable methods.

5.2.2 Local Event Detector Issues

In Figure 3, the University of Florida's Local Event Detector (LED) is composed of three main components: rule, condition/action evaluation, and primitive event detection. The primitive event detector parses events and looks for ECA server primitive events. These events are passed to ECA rules, which are composed of event trees consisting of primitive or composite events, a condition, and an action. Finally, as composite events are detected conditions, and possibly actions are evaluated.

There were several issues encountered when making the LED component multithread safe. The most important issue was the granularity of locking throughout the LED code. Initially, we

---

[5] http://www.cs.wustl.edu/~schmidt/ACE.html

[6] Both thread models are supported in the ACE Framework.

selected a lock granularity that corresponds with object instances. The primary reason for this choice was to compute the impact of granularity on the performance of the ECA server. If the costs were prohibitive, then we could easily scale back the level of lock granularity. Although the insertion of locking at the object level was successful, it turns out that because of limitations in the underlying object-oriented database (i.e. no more than one pre-emptive thread could be in a database transaction at any time), we were forced to implement a locking at a coarser granularity.

One issue converting the LED to be multi-thread safe. We modified the LED in several ways to support multiple threads. First, most LED methods were modified to make them re-entrant. Shared Objects were rewritten to provide monitor synchronization.

6   Lessons Learned and Future Directions

This section highlights areas where future work should concentrate to resolve issues related to the design and implementation of the ECA server. These areas are separated into two parts; the lessons learned from our current implementation and the areas those lessons shed light on for future investigation and implementation.

6.1   Lessons Learned

Table 8 lists the highlights of the lessons learned from the design and implementation of the ECA server.

Lessons	Descriptions
A generic CORBA ECA Service is worthwhile.	A generic CORBA ECA Service can provide active processing across multiple unrelated sources of data in the CORBA environment.
Current ECA design is applicable to OSF's Distributed Computing Environment (DCE).	The current ECA server design and implementation uses nothing specific to the CORBA architecture that would preclude building a DCE version of the ECA service.
Multithreading the ECA Server provides many benefits.	The Multithreaded implementation of the ECA server improves performance and simplifies application developer's code.
ECA Wide-Area Network (WAN) properties unknown.	Our approach has not been tested in this environment.

37

ECA and Event Notification interaction must be improved.	The way that the ECA server receives and sends events is more complicated than it should be. This needs to be simplified to improve ease-of-use for application developers.
ECA server infrastructure is flooded by events from the ECA server.	The ECA infrastructure must be smarter about what events it registers for instead of automatically registering for all events.

6.2 Future Directions

Table 8 -- ECA Lessons Learned

This section indicates future work that would enhance the services provided by the ECA server. Some enhancements are inspired by the lessons learned above. Others are derived from work that was not completed because of a lack of time, or there was not sufficient information at the time to make a design choice. Table 9 below presents a list of possible future work.

Direction	Description
Integrate ECA service with CORBA Transaction Service.	The ECA Server was originally associated with an underlying database. However, to make it a generic ECA service it was separated from its underlying database (except, of course, its rule database). By integrating the ECA with the CORBA Transaction Service ECA actions can be rolled back, if specified by the ECA evaluation context.
Integrate the Univ. of Florida's Rule Scheduling component.	Rule scheduling was omitted in the initial implementation due to time constraints. To reach the full potential of the ECA server, rule scheduling should be integrated into the ECA server.
Incorporate ECA Visualization Tools	Visualization tools make it easier to construct and debug ECA rule sets.

	Visualization tools were omitted from the initial implementation due to time constraints.
Improve ECA availability and reliability	Explore the use of a replicated ECA service and persistent Events to increase the availability and reliability of the ECA Server.
Refine Multi-threaded ECA Model	Determine the best model to use in the ECA server and implement it. Our lessons learned with the current multi-threaded implementation will aid the refinement.

Table 9 -- Future ECA directions

Appendix B: Event Condition Action Service

Extensions of the ECA Architecture and Services

Susan Fichera Banks

1   Introduction

The ECA Service architecture combines the technology of the Local Event Detector (LED), the grammar and syntax of the Event-Condition-Action (ECA) schema, and the power of a distributed, Object Request Broker(ORB)-based, object architecture to support active monitoring of, and reaction to, events of critical interest. Building on this service architecture, we extend the client interface to provide flexible functionality for diverse domain needs. The development areas we pursue are selected to support the use of ECA Services through graphical tools and well-known mechanisms of event communication. Our TIEs within the JFACC program group demonstrate our success in achieving our design goals.

We focus on four areas which are important to our success as a user service: client support methods in the ECA Server interface, the persistence of ECA Rules beyond the Server process, architectural extensions allowing special-purpose code to be executed as a plugin, and support for a variety of communication paths by which clients can receive information from the Server. We build client programs which test these areas and exercise the infrastructure that supports distributed notifications.

1.1   Client Interactions with the ECA Server

We develop client tools and programs to exercise features of the ECA Server, and create database content for the PlanServer and ECA Server. Our client programs define basic events, conditions and actions, and general ECA rules related to missions, units and targets. Notification of the firing of these rules can be broadcast on project-common event channels, such as those provided by JCS and Visualization. We also create our own event channels specifically for weather report broadcasts.

1.1.1 A TIE with METOC Weather Services

A client application may well be interested in weather when planning for actions that will be conducted in various parts of the world. We pursued a TIE with METOC Weather Services to provide weather reports that were temporally and geographically related to specific JFACC

missions, and to deliver those reports to clients that requested weather information for those specific areas.

To create an environment in which to demonstrate the capabilities of our system, we build a plan database, populate plans with objects, and initiate triggers on the objects which can then be processed by the ECA Server. We create a catalog of WeatherRule objects in the PlanServer, to facilitate interaction with the METOC Server. WeatherRule objects are used to pass requests for weather to METOC, and to receive weather evaluations from METOC. ECA Rules are created on Weather Rule objects in the plan; the ECA Server receives triggers when METOC updates the objects with a weather evaluation. Condition evaluations in the ECA Server assess the impact of weather on defined missions in the plan. The resulting ECA action may be to broadcast a weather report on an event channel, notify individual listeners via triggers, or initiate a workflow process. Client applications can choose to listen for weather that relates specifically to their missions or areas of interest.

1.1.2 Notification Services

An objective in our development of client applications is evaluation of methods of notification delivery from ECA Server to clients. We have worked with the Universal Trigger Architecture (UTA), the Orbix NameService, and Orbix Event Service, individually and in combination. The UTA is the simplest interface to use, requiring the instantiation of a trigger object and the implementation of a callback. The UTA is part of the JTF ATD Architecture, and requires a Trigger Server to maintain a table of all registered triggers. The UTA is used for notifications between the PlanServer and ECA Server. It is also an option for notification delivery between ECA Server and its clients.

The API for Orbix NameService is more cumbersome than UTA, both on the server and client side. NameServer and UTA were used in combination, as an alternative way for the ECA Server to locate the trigger communication endpoint without relying on a Trigger Server. A client installs its trigger object in a known part of the NameService namespace; the ECA Server invokes the callback method on all trigger objects installed in that part of the namespace.

Orbix Event Service provides a broadcast protocol for delivery of events. An Orbix-specific API for Talker and Listener is easier to use; the CORBA-standard API is more complex. The ECA Server does not need to deliver a notification to each registered listener individually when using event channels, because the producer and consumer of an event are decoupled by the event channel API.

A single-threaded C++ client enters a loop to poll for triggers or events. A Java client can create a separate thread for this polling, while performing other work. Clients reliably received notifications from the ECA Server with all three delivery mechanisms. The time to deliver is impacted by the volume of event traffic in the system, the overhead of CORBA communications, and general activity level of the system.

2   Problems with the JFACC Architecture

We encountered various problems with the JFACC Architecture which required work-arounds when executing our programs in that environment. Symbol conflicts between core c2schema and JFACC extension idl made it impossible to compile the two schema together. This required temporary modification of the idl, which would have caused us problems in integration with other JFACC applications down the line.

There was no Plan Catalog type defined in the JFACC schema. Thus, there was no API to identify a plan by name, to get a list of plans in the database, or to associate objects with plans and find them again. The only way to find objects in the database was by object reference.

The handling of alerts and events in JFACC was not properly designed. JFACC idl was not compiled with support for typecodes. As a result, schema objects could not be converted to specific types and passed on event channels. The object which instigated the firing of a rule can only be referenced, rather than passed directly. The overhead required to get around this limitation makes event handling in the JFACC environment inefficient and slow. Two objects are created to represent the event: a JfaccEvent and a JfaccOperatorAlert. The JfaccOperatorAlert contains the stringified reference to the plan object causing the alert. The stringified reference of the JfaccOperatorAlert is contained in the JfaccEvent. The stringified reference of the JfaccEvent is passed on the event channel to interested listeners. Both Jfacc objects are stored in the plan database. When event traffic is high, the plan database grows quickly to an unmanageable size due to the storage of event objects.

Clients receiving an event must perform four CORBA invocations to retrieve the subject of the event: the consumption of the event from the channel, the retrieval of the JfaccEvent from the plan, the retrieval of the JfaccOperatorAlert from the plan, and the retrieval of the plan object which triggered the event. The retrieval of objects from the plan is achieved by invocations of the get_graph() method on the PlanServer interface. The impact on a single-threaded PlanServer that must satisfy these multiple invocations across many events is incapacitating.

3   Lessons Learned

In dealing with event channels, we observe that channel parameters, such as queue size and event timeout, have to be adjusted to accommodate the large volume of traffic that pass through a large project environment. Orbix Event Service has many configuration parameters for event channels that can be tuned for better performance. Creation of individual event channels to broadcast events of a particular subject or interest is preferable to a single alert channel that carries all traffic. Special-interest channels could be published like a broadcast guide, possibly through the Orbix NameService.

As stated previously, the UTA trigger has the simplest API for a programmer, and the Orbix CORBA services have more complicated APIs along with greater flexibility and utility. Selection of a notification mechanism would depend on the anticipated volume of notification traffic and the number of potential listeners. Event channels would be a better choice for large-scale broadcasts of alert notifications.

In using event channels as a medium of communication by the ECA Server, we learned that event channel descriptors must be correctly managed in repeatedly-executed code to prevent hangups on channels. Descriptors must be declared external to functions and explicitly freed on function exit. Disconnects from the channel must also be done explicitly through the API. We see a need for default functions in our server which standardize the use of event channels in user-supplied condition and action functions.

4   Work in Progress

Tasking we continue to pursue includes: validation procedures for plugin libraries; separate threads of execution for condition and action functions; parameter passing between conditions and actions to share evaluation results; persistence of event occurrences in the LED; the ability to define rules on event instances as well as event classes; the ability to evaluate gridded weather data and correlate with geolocations of plan elements.

Appendix C: Alert Service

HCI for the ECA Services

David Perham

1 Goal

The goal of this effort was to produce a Human Computer Interface which would provide users with the means to create and maintain a database of ECA Weather Rules. The primary requirement was to have an interface in place in time for use at IFD2.0 in May, 1998.

There were three major areas of functionality required of the interface:

1. Creation of ECA Weather Rules,
2. Creation of ECA Monitor Conditions,
3. Maintenance of the ECA Weather Rules Database.

There were two primary constraints driving the design of the interface:

4. CORBA as the middleware between the HCI and the ECA Weather Rule Server,
5. Objectivity as the Object Oriented Database for the ECA Weather Rule Server,

The HCI was to be developed on a Sun Unix paltform, but the final product was required to be platform independent.

2 Potential Courses of Action

Two courses of action were considered to create the HCI. The first course of action was to create a specialized interface from scratch. The alternative was to build the HCI using ISI's adaptive forms toolkit. There were several factors which impacted the choice between these two courses of action, including:

1. 1. Effort required to complete first, an HCI suitable for IFD2.0, and secondly, one suitable for "real world" use.
2. 2. Desire to use other COTS or GOTS products, wherever possible.
3. 3. Platform independence.

Of course, each of the areas of functionality and design constraints detailed in Section I were also important considerations when choosing the HCI design approach.

For Integration Feasibility Demonstration(IFD) 2.0 it was decided to create the HCI using the ISI adaptive forms toolkit.

3   Reasons For Choosing Adaptive Forms (Advantages)

Adaptive Forms was chosen as the basis for the HCI because of the desire to demonstrate a collaborative use of another JFACC product. The primary short-term advantage of choosing adaptive forms in the design and coding of the HCI for the purpose of IFD2.0 was the ability to create platform independent primitive dialog windows for the creation of ECA Weather Rules and Monitor Conditions relatively quickly. An additional factor was the interest ISI had in seeing us use their product, and their promise of technical support.

4   Reasons For Choosing To Create A Specialized Interface

There are several disadvantages to using Adaptive Forms as the basis of the HCI. The primary disadvantage is that Adaptive Forms only provides a one-way interface, there is no way to display program feedback to the user using the Adaptive Forms Toolkit. One of the purposes of the HCI is to allow users access to a library of database maintenance routines, some or all of which may return data of interest to the user (a list of available ECA Rules, for example). For this reason, Adaptive Forms lack of a two way interface is a major impediment to the creation of a cohesive HCI. Several features of the Adaptive Forms interface are not used by the ECA Weather Rules HCI. These graphical features remain part of the interface in the form of disabled (browned out) buttons. The presence of these buttons caused confusion among the integration test team. Traversing the Adaptive Forms Interface to create Weather Rules and Monitor Conditions was forced and counter-intuitive, another cause for confusion among users (integration testers in this case).

Configuration management is another area wherein the disadvantages of using Adaptive Forms become apparent. New versions of Adaptive Forms may or may not be backwards compatible; incorporating the latest version of the Interface may require code changes within the ECA Weather Rules HCI. The size of the program may be increased beyond a reasonable amount if newer (or just different) versions of java and/or java toolkits (i.e. Swing) are required for Adaptive Forms than are currently used by the ECA Weather Rules HCI. This problem manifested itself when an attempt was made to build a demo of the interface using the ECA Weather Rules software created during IFD2.0. The version of Adaptive Forms which was used in IFD2.0 was now obsolete, and would not build

with any current version of the Swing graphical toolkit available from Sun or other sources. In order to get the interface working, the new version of Adaptive Forms had to be imported, ECA Weather Rule HCI code modified, and a newer version of the Swing toolkit had to be acquired. It should be noted that the POC for Adaptive Forms, Martin Franks, was very quick to lend his expertise to this problem.

5   Interface Described

As the first step in the development of the HCI, an Adaptive Forms grammar file had to be created. Because this document is not intended to be an extensive discussion of the workings of Adaptive Forms, the grammar file is attached as Appendix A. Briefly described, all data which is presented within the ECA Weather Rules Adaptive Forms Dialog is generated from text strings contained in the grammar file; all Weather Rule or Monitor Condition data elements presented for selection by the Adaptive Forms user are also taken from text strings contained in the grammar file. These text strings are returned to the user program (the ECA Weather Rules HCI) in the form of a "tree" of textual data which had to be parsed and converted as appropriate to create data elements appropriate to either an ECA Weather Rule or Monitor Condition. The Adaptive Form presented Database maintenance commands (i.e. List ECA Rules) that were also parsed out of this tree, and executed directly via appropriate system shell scripts. Although the long term goal is platform independence, for the purposes of IFD2.0, the Database maintenance commands were handled in a platform dependent manner, specifically, the HCI used the unix "System" command to execute unix shell commands. Each time the Adaptive Form "Apply" button was clicked, a single ECA Weather Rule or Monitor Condition was created and added to the ECA Weather Rule Database, or a Database Maintainance shell command script was executed. Information returned from execution of these scripts was presented in a java awt dialog.

In order to create ECA Weather Rules, the ECA Weather Rule Server HCI had to take the data elements returned by the Adaptive Form and convert them to data elements appropriate to ECA Weather Rule objects.

Once the data elements had been converted they were used to populate an ECA Weather Rule object, which was passed to the ECA Weather Rule Server via CORBA. The ECA Weather Rule object was defined in the CORBA IDL.

An ECA Weather Rule created from the Adaptive Form could look like this:

"Create Rule for JFACC Hill AFB precipitation/inches 12 hours becomes MARGINAL when GREATER THAN 04 inches/12 hours and becomes UNFAVORABLE at 08 inches/12 hours."

ECA Weather Rule objects contain the following data elements:

Element Name	Element Type
Weatherstate	_weatherstate; // type string
OperationType	_type; // type string
threshold1	_threshold1; // type double
threshold2	_threshold2; // type double
evcriteria	_evaluation; // type int
type	_unitType; // type string
upcriteria	_wxcriteria; // type int
length	1;
altitude	_altitude; // type int

The example ECA Weather Rule would have the following values assigned to each data element:

Element Name	Example Value
weatherstate:	Precipitation
operationType:	Field Takeoff
threshold1:	04
threshold2:	08
evcriteria:	1 (always 1 for IFD2.0)
type	inches/12 hours
upcriteria:	1
altitude:	
altitude	null (not used for IFD2.0)

Each data element not of type string had to be converted from string to integer, as appropriate. Weatherstate represented the general type of operation this rule pertained to, and had values such as: Precipitation, Cloud Cover (tenths), etc. OperationType was the specific type of the Weather Rule, and had values such as: Precipitation (inches/12 hours), Cloud Cover, etc. The value for operationType were drawn directly from the ECA Weather Rules Adaptive Form, and was the basis for the value assigned to weatherstate. UnitType represented the unit of measurement for the Weather Rule (inches per hour, feet, degrees, etc). Threshold1 and threshold2 represented the values of unitType at which the Weather was deemed "Marginal" (threshold1) or "Unfavorable" (threshold 2).

The values of these fields had to be converted from string, and also required calculation in some cases. An example of this is the temperature as used in the TIE with METOC. METOC always required temperature be in degrees Celsius. In the case of an ECA Weather Rule of operationType "Degrees Fahrenheit," the thresholds had to be converted from Fahrenheit (as chosen by the ECA Weather Rules Adaptive Form User) to Celsius (as required by METOC). The Upcriteria field was always set to 1 for IFD2.0, but was intended as a priority number or processing code. The altitude field was not used in IFD2.0.

A similar procedure was followed for creation of ECA Monitor Conditions, which had the following data elements:

operationType,

DateTimeSequence,

LocationSequence,

"NewAlert"

The OperationType was a string indicating the type of mission the Monitor Condition pertained to (i.e. Conventional Bombing, Anti-Submarine Warfare, etc). This value was drawn directly from a selection made by the user of the ECA Weather Rule Server Adaptive Form.

Each Monitor Condition consisted further of a sequence of date time groups which represented the duration of the Monitoring Condition. At the time of IFD2.0, for the purpose of the METOC TIE, only durations consisting of a start and end time were supported. The HCI also allows for durations to be expressed in other formats, such as "For period x from time y." The start and end times for ECA Monitor Conditions had to

be converted from dd/yy/mm format (as chosen by the ECA Weather Rules Adaptive Forms Dialog user) into Julian format (as required by METOC).

Each Monitor Condition also contained a sequence of one or more Latitude/Longitude pairs which defined the area of the ECA Monitor Condition. For the purposes of IFD2.0, a single point location was all that was required, however the HCI allows for the creation of rectangular regions of interest, which were defined by setting two Lat/Lon pairs. These values had to be translated from their raw values to values useful to the ECA Weather Rule Server. For example, the Adaptive Form generated attitude "50 Degrees South" would be converted to the numeric value -50.0. The string value "New Alert" was hardcoded for IFD2.0, and was meant to categorize the Monitor Object.

The following Database Maintainence options were presented:

Start Weather Eval Server

Cleanup Weather Rule Database

Cleanup ECA Rule Database

List ECA Rules

As described above, these functions were invoked via unix shell scripts, and returned information was presented in java awt dialogs. Although not displayed with the current version of the Adaptive Form, an experimental "Start METOC Server" option was tried out at IFD2.0.

The HCI was written in java. It imported several classes generated by the WSR Weather Rules CORBA IDL. The program itself had these java classes:

RuleCondition

Defined CORBA ECA Weather Rule Object, translated incoming data to populate the Object. Moved the Object to the ECA Weather Rule Server Database via CORBA.

MonitorCondition

Defined CORBA ECA Weather Monitor Condition Object, translated incoming data to populate the Object. Moved the Object to the ECA Weather Rule Server Database via CORBA.

SimpleIntegrated

Main driver, brought up and maintained the Adaptive Form, and presented return data in java awt dialogs. Invoked Rule of Monitor constructors, or unix shell scripts to execute Database Maintainence functions.

6   Alert Monitor Interface

The Alert Monitor Interface was created to demonstrate the ability to notify interested clients of incoming alert data. This interface was a platform independent java application, developed on an NT box. The Server periodically polled file data which was then pushed to interested Clients. Clients defined their interest by way of command line arguments at program start up. Data was transferred via sockets, with "I am here" notifications being passed from Client to Server, and Alert Notifications being passed from Server to Client. The Alert Notifications consisted of text strings, containing world wide web addresses of interest to the Client. Upon receipt of an Alert Notification, the Client would visit the specified web site and load the data into a simple browser window. The Web Server hosting the Weather Data used for the demo of this application was running Web Site software.

Apendix C Annex 1: The ECA Weather Rules Adaptive Forms Grammar File

```
// Regular expressions for user input.

UserDefinedRole ([a-zA-Z][a-zA-Z ]+) "Custom Role"

Cardinal ([0-9]+) "Positive Number"

ArbitraryText () "Free-form Text"


// First symbol is the starting symbol.

z "WeatherRule Server": ecaCommand

    ;

////////////////////////////////////////////////////////////////
/////

//Top Level Actions

////////////////////////////////////////////////////////////////
/////

ecaCommand :

    'Create Rule' ruleWhat

    | 'Monitor Weather Condition' monitorWX

    | 'Start Weather Eval Server' starteval

    | 'Cleanup Weather Rule Database' runCleanWR

    | 'Cleanup ECA Rule Database' runCleanECA

    | 'List ECA Rules'

    | 'Run Demo' runDemo

    ;

    //| 'Edit' listWhat

    //| 'List' listWhat
```

```
//| 'Monitor Objects' monitorObj


///////////////////////////////////////////////////////////////

//    runCleanWR

///////////////////////////////////////////////////////////////

runCleanWR

    "Confirm, delete all Weather Rules in the database?" :

    'Confirm, delete all Weather Rules in the database?'
confirm

    ;


///////////////////////////////////////////////////////////////

//    runCleanECA

///////////////////////////////////////////////////////////////

runCleanECA

    "Confirm, delete all ECA Rules in the database?" :

    'Confirm, delete all ECA Rules in the database?' confirm

    ;



///////////////////////////////////////////////////////////////

//    runDemo

///////////////////////////////////////////////////////////////

runDemo

"Run Demo" :

'Confirm, Run Demo' confirm ;
```

```
//////////////////////////////////////////////////////////////

//   starteval

//////////////////////////////////////////////////////////////

starteval

     "Start Eval Server" :

     'Start Eval Server' confirm

;




//////////////////////////////////////////////////////////////

// confirm

//////////////////////////////////////////////////////////////




confirm -hide : 'YES' | 'NO';




//////////////////////////////////////////////////////////////

// Weather Rule Conditions

//////////////////////////////////////////////////////////////

ruleWhat :

     'Weather Rule Conditions' defineWXRC ;

     //'Weather Rule Conditions' defineWXRC |

     //'Event-Condition-Action (ECA) Rules triggered by' event
'Evaluate'

//conditions 'Complete' actions;
```

```
//Listing of Rules

//listWhat -hide: 'Weather Rule Conditions' listWXRC |

//'Event-Condition-Action (ECA) Rules' listECARules;


//listWXRC -hide: 'for' operationsMonitor;


//listECARules -hide: 'currently defined for' ecaRules;


//ecaRules -hide:

    //'Evaluating Climatology' |

    //'Monitoring Mission Weather' |

    //'Monitoring Area Weather'

    //;


/////////////////////////////////////////////////////////////////

//Monitoring Weather Conditions for locations, areas, and time
or time periods
/////////////////////////////////////////////////////////////////

monitorWX -hide: 'for'

operationsMonitor |

location

'during the period' periodTime

;


/////////////////////////////////////////////////////////////////

//Monitoring Objects in the Plan Server

/////////////////////////////////////////////////////////////////
```

```
//monitorObj -hide : event 'of Type' objectsList

      //'evaluate the Condition function' assignCondition

      //'execute the Action function' assignActions

      //'during the period ' periodTime | ;


//////////////////////////////////////////////////////////////

//types of Objects that can be monitored

//////////////////////////////////////////////////////////////


//objectsList -hide :

      //'Objective' |

      //'Task' |

      //'Activity' |

      //'Mission';

//////////////////////////////////////////////////////////////

//Creating RuleConditions

//////////////////////////////////////////////////////////////


//RuleCondition->operationsType

//All Air Operations =

operationsMonitor "Type of Operations" :

      'ALL Operations' |

      'ALL Air Operations' |

      'ALL Strike Operations' |

      'ALL Naval Operations' |

      'ALL Intell Operations' |

      'ALL Ground Operations' |
```

```
'Hill AFB' |

'Conventional Bombing - MidLevel' | //Air Ops, Strike Ops

'Conventional Bombing - LowLevel' | //Air Ops, Strike Ops

'Night Vision - Air' | //Air Ops

'Drones' | //Air Ops, Intell Ops

'Reconnaisance - High' | //Air Ops, Intell Ops

'Reconnaisance - Low' | //Air Ops, Intell Ops

'Reconnaisance - Ground' | //Air Ops, Intell Ops

'Aerial Refueling' | //Air Ops

'Tactical AirLift' | //Air Ops

'Infared Systems' | //Air Ops

'Predator' | //Air Ops, Intell Ops

'Combat Surface Support - Air' | //Air Ops

'Helicopter Ops' | //Air Ops

'Attack Helicopter Ops' | //Air Ops

'Close Air SupportAir Interdiction' | //Air Ops

'Electro-Optical & Night Vision' | //Air Ops

'Intel: Electronic' | //Air Ops, Intell Ops

'Intel: Reconnaisance' | //Air Ops, Intell Ops

'Flight Ops' | //Air Ops

'Naval Refueling' | //Naval Ops

'Carrier Flight Ops' | //Naval Ops, Air Ops, Strike Ops

'Anti-Ship Missile Ops' | //Naval Ops

'Anti-Submarine Warfare' | //Naval Ops

'SeaPort Point of Departure' | //Naval Ops

'AirPort Point of Departure' | //Naval Ops

'Air-to-Air Visual' | //Naval Ops, Air Ops
```

```
'Patrol Boat Ops' | //Naval Ops

'Amphibious Ops' | //Naval Ops

'Amphibious Ops - Air Assault' | //Naval Ops

'Amphibious Ops - Landing Craft' | //Naval Ops

'Anti-Surface Over-the-Horizon' | //Naval Ops, Strike Ops

'Combat Surface Support - Naval' | //Naval Ops

'Minesweeper Ops' | //Naval Ops

'Minesweeper Ops Aviation' | //Naval Ops

'Minesweeper Ops Hunt' | //Naval Ops

'Minesweeper Ops Sweep' | //Naval Ops

'Minesweeper Ops EOD Divers' //Naval Ops

'NBC Chemical' | //Ground Ops

'NBC Smoke' | //Ground Ops

'Personnel Land' | //Ground Ops

'Personnel Airborne' | //Ground Ops

'Forward Arming and Refueling' | //Ground Ops

'Cross Country Maneuvers' | //Ground Ops

'Bridging' | //Ground Ops

'Armor Gun Sighting' | //Ground Ops

'TOW Missiles' | //Ground Ops

'Helicopter Ops - Ground' | //Ground Ops

'Hellfire Missiles' | //Ground Ops

'LOBL Lock-On Before Launch' | //Ground Ops

'LOAL Lock-On After Launch' | //Ground Ops

'CopperHead Missiles' | //Ground Ops

'Air Support - Ground' | //Ground Ops

'ParaDrop' | //Ground Ops
```

```
    'Artillery' | //Ground Ops

    'Combat Surface Support - Ground' | //Ground Ops

    'Air Defense' //Ground Ops

    ;


operationsTypes "Type of Operations" :

    'Hill AFB' |

    'Conventional Bombing - MidLevel' | //Air Ops, Strike Ops

    'Conventional Bombing - LowLevel' | //Air Ops, Strike Ops

    'Night Vision - Air' | //Air Ops

    'Drones' | //Air Ops, Intell Ops

    'Reconnaisance - High' | //Air Ops, Intell Ops

    'Reconnaisance - Low' | //Air Ops, Intell Ops

    'Reconnaisance - Ground' | //Air Ops, Intell Ops

    'Aerial Refueling' | //Air Ops

    'Tactical AirLift' | //Air Ops

    'Infared Systems' | //Air Ops

    'Predator' | //Air Ops, Intell Ops

    'Combat Surface Support - Air' | //Air Ops

    'Helicopter Ops - Air' | //Air Ops

    'Attack Helicopter Ops - Air' | //Air Ops

    'Close Air Support & Air Interdiction' | //Air Ops

    'Electro-Optical & Night Vision' | //Air Ops

    'Intel: Electronic' | //Air Ops, Intell Ops

    'Intel: Reconnaisance' | //Air Ops, Intell Ops

    'Flight Ops' | //Air Ops

    'Naval Refueling' | //Naval Ops
```

'Carrier Flight Ops' | //Naval Ops, Air Ops, Strike Ops

'Anti-Ship Missile Ops' | //Naval Ops

'Anti-Submarine Warfare' | //Naval Ops

'SeaPort Point of Departure' | //Naval Ops

'AirPort Point of Departure' | //Naval Ops

'Air-to-Air Visual' | //Naval Ops, Air Ops

'Patrol Boat Ops' | //Naval Ops

'Amphibious Ops' | //Naval Ops

'Amphibious Ops - Air Assault' | //Naval Ops

'Amphibious Ops - Landing Craft' | //Naval Ops

'Anti-Surface Over-the-Horizon' | //Naval Ops, Strike Ops

'Combat Surface Support - Naval' | //Naval Ops

'Minesweeper Ops' | //Naval Ops

'Minesweeper Ops Aviation' | //Naval Ops

'Minesweeper Ops Hunt' | //Naval Ops

'Minesweeper Ops Sweep' | //Naval Ops

'Minesweeper Ops EOD Divers' //Naval Ops

'NBC Chemical' | //Ground Ops

'NBC Smoke' | //Ground Ops

'Personnel Land' | //Ground Ops

'Personnel Airborne' | //Ground Ops

'Forward Arming and Refueling' | //Ground Ops

'Cross Country Maneuvers' | //Ground Ops

'Bridging' | //Ground Ops

'Armor Gun Sighting' | //Ground Ops

'TOW Missiles' | //Ground Ops

'Helicopter Ops - Ground' | //Ground Ops

```
'Hellfire Missiles' | //Ground Ops

'LOBL Lock-On Before Launch' | //Ground Ops

'LOAL Lock-On After Launch' | //Ground Ops

'CopperHead Missiles' | //Ground Ops

'Air Support - Ground' | //Ground Ops

'ParaDrop' | //Ground Ops

'Artillery' | //Ground Ops

'Combat Surface Support - Ground' | //Ground Ops

'Air Defense' //Ground Ops

;



//RuleCondition->owner = role

//RuleCondition->comment = setDescription

//RuleCondition->operationsType = operationsTypes

//RuleCondition->weatherState = weatherList

defineWXRC "User Role?" :

    'for' role

    'to monitor weather for' operationsTypes

    'when' weatherList

    'Description:' setDescription

    ;



//RuleConditi0n->weatherstate

weatherList :

    'Visibility {100s of ft}' feet100 |

    'Surface Visibility {100s of ft}' feet100 |
```

```
'Precipitation {inches/12 hours}' inperhr |

'Cloud Cover {eigths}' eigths |

'Cloud Cover {tenths}' tenths |

'Precipitation {levels}' levels |

'Fog/Obscuration {level}' fog |

'Wind Speed {knots}' knots |

'Wind Speed at Altitude {knots}' altitude knots |

'Head Winds Speed {knots}' knots |

'Cross Winds Speed {knots}' knots |

'Surface Wind Speed {knots}' knots |

'Ceiling {100s of ft agl}' feet100 |

'Humidity {percent}' percent|

'Absolute Humidity {g/m3}' humidity |

'Temperature {degree F}' degF |

'Temperature {degree C}' degC |

'Temperature at altitude {degree C}' altitude degC |

'Lightning {Distance in nm}' shortdistance |

'ThunderStorms {Distance in nm}' meddistance |

'ThunderStorms {levels}' leveltsms |

'Ground Conditions {Moisture}' groundcond |

'Snow Accumulation {inches}' inches |

'Icing {levels}' levels |

'Turbulence {levels}' levels |

'Moonlight' moon |

'Transmittance' transmittance |

'Illumination {Foot Candles}' illumination |

'Wave Height {feet}' feet |
```

```
        'Combined Seas' feet |

        'Breaker Height' feet |

        'Wave Period' seconds |

        'Current' knots

        ;


//RuleCondition->wxcriteria

criteria -hide : 'when LESS THAN' | 'when GREATER THAN';

criterialess -hide : 'when LESS THAN';

criteriagreater -hide : 'when GREATER THAN';


feet "feet" : 'becomes MARGINAL' criteria numbers0to20 'ft and
becomes UNFAVORABLE at' numbers0to20 'ft';


feet100 "100s of feet" : 'becomes MARGINAL' criterialess
numbers0to90 '00 ft and becomes UNFAVORABLE at' numbers0to90 '00
ft';


seconds "Seconds" : 'becomes MARGINAL' criteria numbers0to20 'ft
and becomes UNFAVORABLE at' numbers0to20 'ft';


altitude "1000s of feet" : 'at flight altitude of' numbers0to90
'000 ft';


knots "Knots" : 'becomes MARGINAL' criteriagreater numbers0to90
'kts and becomes UNFAVORABLE at' numbers0to90 'kts';
```

```
tenths "Tenths of Cloud Cover" : 'becomes MARGINAL' criteria
numbers0to10 'tenths and becomes UNFAVORABLE at' numbers0to10
'tenths';


eigths "Eigths of Cloud Cover" : 'becomes MARGINAL' criteria
numbers0to8 'eigths and becomes UNFAVORABLE at' numbers0to8
'eigths';


percent "Percent" : 'becomes MARGINAL' criteria tens0to100
'percent and becomes UNFAVORABLE at' tens0to100 'percent';


degF "Degrees Farenheit" : 'becomes MARGINAL' criteria
plusorminus numbers0to90 'deg F and becomes UNFAVORABLE at'
plusorminus numbers0to180 'deg F';


degC "Degrees Celsius" : 'becomes MARGINAL' criteria plusorminus
numbers0to90 'deg C and becomes UNFAVORABLE at' plusorminus
numbers0to180 'deg C';


inperhr "Inches per 12 Hours" : 'becomes MARGINAL'
criteriagreater numbers0to20 'inches/12 hours and becomes
UNFAVORABLE at' numbers0to20 'inches/12 hours';


inches "Inches Accumulation" : 'becomes MARGINAL' criteria
numbers0to20 'inches/hour and becomes UNFAVORABLE at'
numbers0to20 'inches/hour';


levellist "Observed to be" : 'NONE' | 'LIGHT' | 'MODERATE' |
'HEAYY/SEVERE';
```

```
levels -hide : 'becomes MARGINAL when' levellist 'and becomes
UNFAVORABLE when' levellist;


fog -hide : 'becomes MARGINAL when' levellist 'and becomes
UNFAVORABLE when' levellist;


leveltsmslist "Observed to be" : 'NONE' | 'LIGHT/SCATTERED' |
'MODERATE/FEW' | 'HEAYY/SEVERE';


leveltsms -hide : 'becomes MARGINAL when' leveltsmslist 'and
becomes UNFAVORABLE when' leveltsmslist;


groundcondlist "Observed to be" : 'DRY' | 'MOIST' | 'WET';


groundcond -hide : 'becomes MARGINAL when'groundcondlist 'and
becomes UNFAVORABLE when' groundcondlist;


shortdistance "Nautical Miles" : 'becomes MARGINAL' criteria
decimals0to5 'nm from the site and becomes UNFAVORABLE at'
decimals0to5 'nm from the site' ;


meddistance "Nautical Miles" : 'becomes MARGINAL' criteria
numbers0to90 'nm from the site and becomes UNFAVORABLE at'
numbers0to90 'nm from the site' ;


humidity "Grams per Meter3" : 'becomes MARGINAL' criteria
numbers0to20 'g/m3 and becomes UNFAVORABLE at' numbers0to20
'g/m3';
```

```
transmittance -hide : 'becomes MARGINAL' criteria decimals0topt9
'and becomes UNFAVORABLE at' decimals0topt9;


illumination "Foot Candles" : 'becomes MARGINAL' criteria
numbers0to20 'ft candles and becomes UNFAVORABLE at'
numbers0to20 'ft candles';


moonlist : 'FULL MOON' | 'MOONRISE' | 'NO MOON';


moon "Level of Moonlight" : 'becomes MARGINAL' moonlist 'and
becomes UNFAVORABLE at' moonlist;


plusorminus -hide : 'PLUS' | 'MINUS';



setDescription "Rule Description": ArbitraryText;



////////////////////
// General Definitions
////////////////////

role:
    'JFACC' |
    'Force Support Planner' |
    'Force Application Planner' |
    'ISR Planner' |
```

```
    'OSA Planner' |

    UserDefinedRole

    ;


/////////////////////

// Time

/////////////////////

timeInterval -hide : 'DAYS' | 'HOURS' | 'MINUTES';


periodTime "Period" : startTime endTime;


startTime -hide "Starting Time" : 'from' dayAndTime | 'starting
now' | 'starting in' numbers0to90 hours 'from now';

endTime -hide "Ending Time" : 'to' dayAndTime | 'ending'
numbers0to90 timeInterval 'from start time';


time "Time in 24 Hr Clock": hours ;

//timeHHMM "Time": hoursmin ':' minutes ;


hours "Hours":

    '0000' | '0100' | '0200' | '0300' | '0400' | '0500' |
'0600' | '0700' | '0800' | '0900' | '1000' |

    '1100' | '1200' | '1300' | '1400' | '1500' | '1600' |
'1700' | '1800' | '1900' | '2000' |

    '2100' | '2200' | '2300' ;



//hoursmin "Hours":
```

```
    //'1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' |
'10' |
    //'11' | '12' | '13' | '14' | '15' | '16' | '17' | '18' |
'19' |
//'20' |
    //'21' | '22' | '23' ;


minutes "Minutes":
    '00' | '01' | '02' | '03' | '04' | '05' | '06' | '07' |
'08' |
'09' | '10' |
    '11' | '12' | '13' | '14' | '15' | '16' | '17' | '18' |
'19' |
'20' |
    '21' | '22' | '23' | '24' | '25' | '26' | '27' | '28' |
'29' |
'30' |
    '31' | '32' | '33' | '34' | '35' | '36' | '37' | '38' |
'39' |
'40' |
    '41' | '42' | '43' | '44' | '45' | '46' | '47' | '48' |
'49' |
'50' |
    '51' | '52' | '53' | '54' | '55' | '56' | '57' | '58' |
'59' ;


days31 "Day":
```

```
      '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '10'
|
      '11' | '12' | '13' | '14' | '15' | '16' | '17' | '18' |
'19' |
'20' |
      '21' | '22' | '23' | '24' | '25' | '26' | '27' | '28' |
'29' |
'30' |
      '31';


days30 "Day":
      '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '10'
|
      '11' | '12' | '13' | '14' | '15' | '16' | '17' | '18' |
'19' |
'20' |
      '21' | '22' | '23' | '24' | '25' | '26' | '27' | '28' |
'29' |
'30' ;


days28 "Day":
      '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '10'
|
      '11' | '12' | '13' | '14' | '15' | '16' | '17' | '18' |
'19' |
'20' |
      '21' | '22' | '23' | '24' | '25' | '26' | '27' | '28' ;


year "Year":
```

```
    '1997' | '1998' | '1999' | '2000' | '2001' ;


dayAndTime "Time" :

    'Jan' days31 ',' year 'at' time |

    'Feb' days28 ',' year 'at' time |

    'Mar' days31 ',' year 'at' time |

    'Apr' days30 ',' year 'at' time |

    'May' days31 ',' year 'at' time |

    'Jun' days30 ',' year 'at' time |

    'Jul' days31 ',' year 'at' time |

    'Aug' days31 ',' year 'at' time |

    'Sep' days30 ',' year 'at' time |

    'Oct' days31 ',' year 'at' time |

    'Nov' days30 ',' year 'at' time |

    'Dec' days31 ',' year 'at' time;


////////////////////////////
/////Location //////////////////
////////////////////////////
position : latitudeDM longitudeDM;



area : 'Upper Latitude' latitude 'Lower Latitude' latitude 'Left
Longitude' longitude 'Right Longitude' longitude;


location : 'at Position' position | 'in the Area Defined by'
area;
```

```
////Latitude and Longitude defined by Degrees & Minutes
latitudeDM "Latitude" :

    'Degrees' numbers0to90 'Minutes' minutes latDir;
longitudeDM "Longitude" :

    'Degrees' numbers0to180 'Minutes' minutes lonDir;


latitude "Latitude in Degrees" :

    numbers0to90 'Degrees' latDir;
longitude "Longitude in Degrees" :

    numbers0to180 'Degrees' lonDir;



latDir "N/S" : 'N' | 'S';
lonDir "E/W" : 'E' | 'W';


tens0to100:

    '00' | '10' | '20' | '30' | '40' | '50' | '60' | '70' |
'80' |
'90' | '100';


numbers0to8:

    '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' ;


numbers0to10:
```

```
    '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' |
'10';


numbers0to20:
    '00' | '01' | '02' | '03' | '04' | '05' | '06' | '07' |
'08' |
'09' | '10' |
    '11' | '12' | '13' | '14' | '15' | '16' | '17' | '18' |
'19' |
'20';


decimals0topt9:
    '0.1' | '0.2' | '0.3' | '0.4' | '0.5' | '0.6' | '0.7' |
'0.8' |
'0.9';


decimals0to5:
    '0.1' | '0.2' | '0.3' | '0.4' | '0.5' | '0.6' | '0.7' |
'0.8' |
'0.9' | '1.0' |
    '1.1' | '1.2' | '1.3' | '1.4' | '1.5' | '1.6' | '1.7' |
'1.8' | '1.9' |
'2.0' |
    '2.1' | '2.2' | '2.3' | '2.4' | '2.5' | '2.6' | '2.7' |
'2.8' | '2.9' |
'3.0' |
    '3.1' | '3.2' | '3.3' | '3.4' | '3.5' | '3.6' | '3.7' |
'3.8' | '3.9' |
'4.0' |
```

```
     '4.1' | '4.2' | '4.3' | '4.4' | '4.5' | '4.6' | '4.7' |
'4.8' | '4.9' |

'5.0';


numbers0to90:
     '00' | '01' | '02' | '03' | '04' | '05' | '06' | '07' |
'08' |

'09' | '10' |
     '11' | '12' | '13' | '14' | '15' | '16' | '17' | '18' |
'19' |

'20' |
     '21' | '22' | '23' | '24' | '25' | '26' | '27' | '28' |
'29' |

'30' |
     '31' | '32' | '33' | '34' | '35' | '36' | '37' | '38' |
'39' |

'40' |
     '41' | '42' | '43' | '44' | '45' | '46' | '47' | '48' |
'49' |

'50' |
     '51' | '52' | '53' | '54' | '55' | '56' | '57' | '58' |
'59'|

'60' |
     '61' | '62' | '63' | '64' | '65' | '66' | '67' | '68' |
'69'|

'70' |
     '71' | '72' | '73' | '74' | '75' | '76' | '77' | '78' |
'79'|

'80' |
```

```
    '81' | '82' | '83' | '84' | '85' | '86' | '87' | '88' |
'89'|
'90' ;


numbers0to180:
    '00' | '01' | '02' | '03' | '04' | '05' | '06' | '07' |
'08' |
'09' | '10' |
    '11' | '12' | '13' | '14' | '15' | '16' | '17' | '18' |
'19' |
'20' |
    '21' | '22' | '23' | '24' | '25' | '26' | '27' | '28' |
'29' |
'30' |
    '31' | '32' | '33' | '34' | '35' | '36' | '37' | '38' |
'39' |
'40' |
    '41' | '42' | '43' | '44' | '45' | '46' | '47' | '48' |
'49' |
'50' |
    '51' | '52' | '53' | '54' | '55' | '56' | '57' | '58' |
'59'|
'60' |
    '61' | '62' | '63' | '64' | '65' | '66' | '67' | '68' |
'69'|
'70' |
    '71' | '72' | '73' | '74' | '75' | '76' | '77' | '78' |
'79'|
'80' |
```

'81' | '82' | '83' | '84' | '85' | '86' | '87' | '88' |
'89'|

'90'

| '91' | '92' | '93' | '94' | '95' | '96' | '97' | '98' |
'99' |

'100' |

'101' | '102' | '103' | '104' | '105' | '106' | '107' |
'108' | '109' |

'110' |

'111' | '112' | '113' | '114' | '115' | '116' | '117' |
'118' | '119' |

'120' |

'121' | '122' | '123' | '124' | '125' | '126' | '127' |
'128' | '129' |

'130' |

'131' | '132' | '133' | '134' | '135' | '136' | '137' |
'138' | '139' |

'140' |

'141' | '142' | '143' | '144' | '145' | '146' | '147' |
'148' | '149' |

'150' |

'151' | '152' | '153' | '154' | '155' | '156' | '157' |
'158' | '159'|

'160' |

'161' | '162' | '163' | '164' | '165' | '166' | '167' |
'168' | '169'|

'170' |

'171' | '172' | '173' | '174' | '175' | '176' | '177' |
'178' | '179'|

```
'180' ;


//event:

    //'the creation of a Plan Server Object' |

    //'the update of a Plan Server Object' |

    //'a periodic event' periodTime |

    //'a aperiodic event' periodTime ;


//assignCondition "Condition Evaluation Functions" :

    //'Evaluate Condition' conditions assignActions;


//assignActions "Actions to be Executed":

    //'Execute action' actions;



//conditions: conditionClauseRequired 'or'
conditionClauseOptional ;


//conditionClauseOptional :

    //conditionClause

    //|

    //;


//conditionClauseRequired :

    //conditionClause

    //;
```

```
//conditionClause: conditionTermRequired 'and'
conditionTermOptional 'and'

//conditionTermOptional ;


//conditionTermOptional :

     //condition

     //|

     //;


//conditionTermRequired :

     //condition

     //;

//

//condition :

     //'Condition 1'

     //| 'Condition 2'

     //;


//actions: actionRequired 'and' actionOptional 'and'
actionOptional ;


//actionOptional :

     //action

     //|

     //;


//actionRequired :
```

```
//action

//;


//action "Action":

//'Action 1'

//| 'Action 2'

//;
```

Appendix D: Information Dominant Decision Environment Demonstration Description

1   Demonstration Objective

The primary objective of this demonstration is to illustrate the capability of active database technology to provide intelligent information management in a real-world environment. There are several obstacles to effective information management in real-world applications, one of the most challenging is relating information across multiple, heterogeneous databases. This demonstration will link information from three distributed real-world data sources: the Air Campaign DB (ACDB) used by TBMCS core for storage of Air Battle Plans and all associated data; real world weather databases; and real world logistics and sustainment databases.

A second major challenge is converting the flood of data updates into useful information for the decision-makers. This can involve intelligent filtering based on user or system defined constraints; relating data updates to existing data to determine criticality; or complex data gathering to provide the decision maker with a more complete understanding of the situation. This demonstration will illustrate examples of both intelligent filtering and complex data gathering.

A third major challenge is minimizing the uncertainty caused by conflicting and inconsistent data. This is becoming a greater and greater issue as technological advances allow users to access a variety of information as if it were contained in a local data source. None of the data sources slated for use in this demonstration have any overlap so there will be no inconsistencies or conflicts between data sources. However, preserving data integrity by preventing data corruption can be demonstrated. This demonstration will include one or more examples of data consistency/integrity rules associated with the ACDB.

2   Data Sources

2.1   Air Operations Data Base

A preliminary review of the database design documents has identified the following tables as possible targets for application of active technology for this demonstration. This list will be further revised and refined once the actual database is imported and the data set can be examined in detail.

2.1.1 Air Battle Plan

ABP	ABP_WW_ID
ABP_Req	ABP_WW_ID

2.1.2 Missions

Msn	Msn_WW_ID
Air_Msn	Msn_WW_ID
Air_Msn_Acft	Air_Msn_Acft_Group_ID

2.1.3 Objectives

AMO	Msn_WW_ID
AMO_ID	
ABP_WW_ID	
AMO_Start	Date
AMO_End	Date
AMO_Priority_ID	
AMO_Sequence_ID	

2.1.4 Joint Tactical Air Strike Request

JTASR	JTASR_ID
ABP	ABP_WW_ID
ABP_Req	ABP_Req_ID
JTASR_Msn_Type_CD	
JTASR_Precedence_CD	

2.2 Weather Server

The Weather Server provides automatic, periodic, reachback capabilities for weather reports and weather forecasts. The weather data is available on both classified and unclassified sites and includes both Air Force and Navy weather databases.

2.2.1 Mission Weather

Monitoring of weather for takeoff location and time, target area and time, and landing location and time.

2.2.2 Area Climatology

Climatology data can be used to determine the probable impact on mission scheduling and weapons effectiveness. Climatology data is reported as monthly averages by reporting station.

2.3 ICIS

Assessment of potential changes in sustainment requirements based on actual operations tempo vice the planned operations tempo. Monitoring of changes in sustainment shortfalls caused by concurrent contingencies.

3 Scenario

The scenario will consist of two phases, the first phase will correlate with the planning phase of a campaign. The second phase will correlate with the continuous planning-execution cycle once an operation commences.

3.1 Planning Phase:

During the actual planning phase, much of the detail available in the ACDB would not be filled in. As parts of the plan are entered into the database e.g. location, resources, automatic process will be fired off to provide climatology analysis, sustainment analysis, etc. For this demonstration, we will simulate the planning phase:

At startup the system will initialize planning monitors (e.g. climatology, sustainment) and initialize mission weather monitoring.

Alerts presented for Climatology for upcoming period.

Alerts presented for Sustainment identifying shortfalls

User will specify one or more ECA rules to be monitored

A series of database updates will be initiated resulting in alerts, which demonstrate the active technology capabilities.

Optional:

Add a new Condition or Action Function. One approach would be to be able to add the new library, update a table in the database with the necessary descriptive information, and have the "dynamic" version of the application automatically include it in the list of available Condition or Action Functions.

3.2   Continuous Planning and Execution Phase

The execution phase will be simulated with a series of update messages to the ACDB which will result in trigger events to be analyzed.

3.2.1 Weather Forecast update

A Weather forecast will be received indicating that some missions will be affected in an upcoming period 12-24 hrs in the future. The impacted missions will be identified, if possible second order impacts will also be identified.

3.2.2 Mission Status

One or more missions will abort or fail with an impact on other related missions. The impacted missions will be identified, if possible second order impacts will also be identified.

3.2.3 Sustainment Assessment

Based on observed operations tempo, the sustainment requirements will be reassessed each day. Operations tempo refers to the number of sorties flown per airframe and the average length of each sortie. The USAF has doctrinal numbers which are used by ICIS to calculate sustainment.

3.2.4 Critical Mobile Target

An urgent Joint Tactical Air Strike Request will be inserted into the ACDB. The user will be alerted. In addition, the location of the target will be displayed on a geographical display along with the current location of both air alert and ground alert missions. Actual locations of air alert missions will be estimated. If possible, additional information will be displayed including: weapons loadout of each mission, air defense capabilities in the vicinity of the target, etc.

3.2.5 User Defined Alerts

One or more user defined alerts will be triggered e.g. monitoring the completion status of a mission or mission package; monitoring of Close Air Support (CAS) requests; etc.

4   New Functionality required

4.1   Oracle Server

The capability to register for and monitor Oracle database CrUD triggers. There is a TBMCS module, which accomplishes this, however if we are unable to obtain this module we will need to create a similar capability.

The capability to query the Oracle database for evaluating Condition and Action functions. The capability to update the Oracle database.

4.2   General Alerting Capability on AODB

4.2.1 Numerical Fields

A Generalized Condition function, which supports numerical constraint, testing through specification of standard arithmetical operators: greater than, less than, equal to, not equal to, between, and not between. The user interface would allow setting of a numerical constraint on any valid numerical field.

4.2.2 DateTime Fields

A Generalized Condition function which supports date constraint testing through specification of standard arithmetical operators. The user interface would allow setting of a data constraint on any valid date field.

4.2.3 Location Fields

A Generalized Condition function, which supports location constraint, testing through specification of standard location comparisons: within area, not within area, within specified range from location (may want to categorize as near, far).

4.3   Mission Alerting

4.3.1 Weather

The capability to establish weather monitoring for all scheduled missions on startup. This is a demonstration initialization procedure. As a minimum, weather should be monitored for takeoff

point, target area, and landing point for the specified time intervals. Monitoring of waypoints, particularly rendezvous points and refueling points would be a nice extension to this capability.

The user should be able to specify monitoring of weather for a location, area, or scheduled mission.

4.3.2 Mission Status

The capability to establish status monitoring for all scheduled missions on startup. This is a demonstration initialization procedure. Notification of the impact of any mission status changes: list of associated/affected missions.

User should be able to specify monitoring of Mission status by type of mission, specific mission, or mission objective.

Optional: Identify lower priority missions which with similar aircraft assignments and weapons loads and similar or later launch times.

4.3.3 New Mission Request

The capability to detect new mission requests and evaluate urgency. Identify location of request on map.

Optional:

1) Plot all airborne and alert missions, which could be assigned to new target on map. Suitable missions would have appropriate weapons capability.

2) Calculate Time-On-Top (TOT) from current location

3) Plot defensive systems in the area of the target.

4.4   Planning

4.4.1 Sustainment

Due to the time to complete an actual ICIS run, all ICIS data runs will probably need to be pre-staged.

The capability to calculate the Force density (Fd) of the Air Battle Plan. Force density is defined as the number of aircraft, by type, by day. The ICIS team will provide the desired data structure or object definition. This is a preplanning procedure, which would actually react to the creation to a new plan and assignment of forces.

The capability to transmit the Fd to the ICIS application and receive a URL providing access to the ICIS analysis.

4.4.2 Monitoring

The capability to monitor numbers of sorties per day, per aircraft type, average flight time per sortie and compare this to default assumptions received from ICIS. The capability to transmit to ICIS the actual operations tempo when it varies from the planned operations tempo and receive a URL providing access to the ICIS analysis.

ICIS will provide the data structure or object definition.

4.4.3 Climatology

The capability to determine the operational area(s) involved in the Air Battle Plan and request appropriate Climatology data from the WX Server.

Appendix E: Air Operations DataBase (AODB) Use Cases

AODB Based ECA Rules

1   Time Critical Target: Urgent Ground Attack Request

Interested Party: Airborne Controller/Attack Planner

1.1 Event:	PLUS (SEQ(Event 1, Event 2), 3 min)
Event 1:	Insert into ABP_REQ
Event 2:	Insert into GRD_ATK_REQ

1.2 Condition:	Comment
ABP_REQ.ABP_REQ_ID = GRD_ATK_REQ.ABP_REQ_ID	
ABP_REQ.ABP_WW_ID = GRD_ATK_REQ.ABP_WW_ID	
ABP_REQ..ABP_REQ_PRIORITY_ID like '1%'	e.g. 1, 1A, 1B, 1C
ABP_REQ_NLT_DTTM > Now()	

1.3   Action:

Scud Sequence of retrievals for map

Need to do some math to determine which defensive assets are an issue

Alert Notify

Priority: HIGH

Text: Ground Attack Request (X min) NLT: ABP_REQ_NLT_DTTM [DDHH24MI MON]

If possible X min calculated as difference between Now() & ABP_REQ_NLT_DTTM

Map Alert

Independent Variables: (TABLE1 ROWID1 TABLE2 ROWID2)

TABLE1.ROWID: Select where ROWID=ROWID1

TABLE2.ROWID: Select where ROWID=ROWID2

2 Time Critical Target: Urgent JTASR Request

Interested Party: Airborne Controller/Attack Planner

2.1 Event:	PLUS (Event 1, Event 2, 3 min)
Event 1:	Insert into ABP_REQ
Event 2:	Insert into JTASR
Event 3:	Insert into CAS_TGT

2.2 Condition:	Comment
ABP_REQ.ABP_REQ_ID = JTASR.ABP_REQ_ID	
ABP_REQ.ABP_WW_ID = JTASR.ABP_WW_ID	
ABP_REQ.ABP_REQ_ID = CAS_TGT.ABP_REQ_ID	
ABP_REQ.ABP_WW_ID = CAS_TGT.ABP_WW_ID	
ABP_REQ..ABP_REQ_PRIORITY_ID like '1%'	e.g. 1, 1A, 1B, 1C
ABP_REQ_NLT_DTTM > Now()	

2.3 Action:

Similar to Scud Sequence of retrievals for map, query CAS_TGT table

Need to do some math to determine which defensive assets are an issue

Alert Notify

    Priority: HIGH

    Text: JATSR Request (X min) NLT: ABP_REQ_NLT_DTTM [DDHH24MI MON]

    If possible X min calculated as difference between Now() & ABP_REQ_NLT_DTTM

Independent Variables: (TABLE1 ROWID1 TABLE2 ROWID2 TABLE3 ROWID3)

    TABLE1.ROWID: Select where ROWID=ROWID1

    TABLE2.ROWID: Select where ROWID=ROWID2

    TABLE3.ROWID: Select where ROWID=ROWID3

Map Alert

3 Weather Monitoring: New Mission

Establish weather monitoring on new mission

3.1 Event:	PLUS (Event 1,)
Event 1:	Insert into AIR_MSN_EVT

3.2 Condition:	Comment
Select MSN_EVENT_REC_TYPE_CD from AIR_MSN_EVT where ROWID = ROWID1	
Select from appropriate table (e.g. LOC_AIR_EVNT, TGT_AIR_EVNT)	

3.3 Action:

Insert Location(s) and Time(s) into Weather Monitoring Grid

Evaluate Wx based on current forecast, alert if required

      Priority: HIGH if WX UNFAVORABLE

          MEDIUM if WX MARGINAL

      Text: MSN_WW_ID Weather Alert Forecast Time [DDHH24MI MON]

Independent Variables: (TABLE1 ROWID1)

      TABLE1.ROWID: Select where ROWID=ROWID1

4 Planning: Sustainment Monitoring

Evaluate sustainment feasibility on new air battle plan

4.1 Event:	
Event 1:	Insert into ABP
Event 2:	Insert into ABP_FORCE_MODULES

4.2 Condition:	Comment
ABP.ABP_WW_ID = ABP_FORCE_MODULES.ABP_WW_ID	

4.3   Action:

Select from ABP_FORCE_MODULES

Evaluate sustainment for each distinct force module.

Alert Notify

>Priority:MEDIUM

>Text: Sustainment Planning Alert: ICIS Analysis Available Plan Name

5   Planning: Climatology

5.1 Event:	
Event 1:	Insert into ABP
Event 2:	Insert into ABP_OPERATIONS_AREA

5.2 Condition:	Comment
ABP.ABP_WW_ID = ABP_OPERATIONS_AREA.ABP_WW_ID	

5.3   Action:

Request Climatology for defined operations area

Evaluate Wx Rules based on Climatology

6   Execution: Sustainment Monitoring

7 Mission Status Change

7.1 Event:	
Event 1:	Update AIR_MSN_STAT

7.2 Condition:	Comment
AIR_MSN_STAT_CD in ('ABT','CHG','CNX','LOS')	Return MSN_WW_ID (could be the result of a query on ROWID?)

7.3 Action:

Select 'define msn-id = 'msn_ww_id from air_msn_stat where ROWID = ROWID1;

select ACFT_MDS_TYPE_CD, ACT_ACFT_MDS_CFG_ID,

AIR_MSN_ACFT_AIRCRAFT_QY

from AIR_MSN_ACFT

where MSN_WW_ID IN (select msn_ww_id from pkg_air_msn

where pkg_id =

 (select pkg_id from pkg_air_msn

where msn_ww_id = '&MSN-ID'))

Alert Notify

      Priority:High if < 3 hrs before planned

          Medium if < 24 hrs before planned

          Low if > 24 hrs before planned

     Text: Mission Status Alert: MM_WW_ID STATUS

Independent Variables: (TABLE1 ROWID1)

      TABLE1.ROWID: Select where ROWID=ROWID1

8   Mission Delay (Estimate)

8.1 Event:	
Event 1:	Update AIR_MSN_EVNT.AIR_MSN_ESTIMATED_DTTM

8.2 Condition:	Comment
AIR_MSN_ESTIMATED_DTTM > AIR_MSN_PLANNED_DTTM + X minutes	Return MSN_WW_ID (could be the result of a query on ROWID)

8.3   Action:

select ACFT_MDS_TYPE_CD, ACT_ACFT_MDS_CFG_ID,

AIR_MSN_ACFT_AIRCRAFT_QY

from AIR_MSN_ACFT

where MSN_WW_ID IN (select msn_ww_id from pkg_air_msn

where pkg_id =

 (select pkg_id from pkg_air_msn

where msn_ww_id = 'RESULT'))

Alert Notify

      Priority:High if < 3 hrs before planned

           Medium if < 24 hrs before planned

           Low if > 24 hrs before planned

     Text: Mission Status Alert: MM_WW_ID STATUS

Independent Variables: (TABLE1 ROWID1)

      TABLE1.ROWID: Select where ROWID=ROWID1

9   Mission Delay (Actual)

9.1 Event:	
Event 1:	Update AIR_MSN_EVNT.AIR_MSN_ACTUAL_DTTM

9.2 Condition:	Comment
AIR_MSN_ACTUAL_DTTM > AIR_MSN_PLANNED_DTTM + X minutes	Return MSN_WW_ID (could be the result of a query on ROWID)

9.3   Action:

select ACFT_MDS_TYPE_CD, ACT_ACFT_MDS_CFG_ID,

AIR_MSN_ACFT_AIRCRAFT_QY

from AIR_MSN_ACFT

where MSN_WW_ID IN (select msn_ww_id from pkg_air_msn

where pkg_id =

 (select pkg_id from pkg_air_msn

where msn_ww_id = 'RESULT'))

Alert Notify

      Priority:High if < 3 hrs before planned

          Medium if < 24 hrs before planned

          Low if > 24 hrs before planned

     Text: Mission Status Alert: MM_WW_ID STATUS

Independent Variables: (TABLE1 ROWID1)

      TABLE1.ROWID: Select where ROWID=ROWID1

10 Weather Monitoring: Mission Update

10.1 Event:	
Event 1:	Update AIR_MSN_EVNT

10.2 Condition:	Comment
AIR_MSN_EVNT_ACTUAL_DTTM OR AIR_MSN_EVNT_PLANNED_DTTM OR AIR_MSN_EVNT_PLANNED_OFF_DTTM Will require table specific trigger which returns Old & New values for each	Determine if time change, changes forecast time window

10.3 Action:

Update WX Mission Grid

Evaluate WX Rules for Mission Grid

Alert as required

Independent Variables: (TABLE1 ROWID1)

      TABLE1.ROWID: Select where ROWID=ROWID1

11 Weather Monitoring: Mission Event Completion

11.1 Event:	
Event 1:	Update AIR_MSN_EVNT

11.2 Condition:	Comment
AIR_MSN_EVNT_ACTUAL_OFF_DTTM NOT NULL	Indicating Mission Event Complete

11.3 Action:

Delete Mission Point from Gridded_Mission

12   Weather Monitoring Mission Status Change

12.1 Event:	
Event 1:	Update AIR_MSN_STAT

12.2 Condition:	Comment
AIR_MSN_STAT_CD in ('ABT',,'CNX','LOS', 'MC','DIV')	Return MSN_WW_ID (could be the result of a query on ROWID?)

12.3 Action:

Delete ALL Mission Points from Gridded_Mission

13   Weather Monitoring: Weather Update

13.1 Event:	
Event 1:	Wx Update received from TEDS

13.2 Condition:	Comment
WX matches monitored grid	

13.3 Action:

Evaluate Weather Rules for Mission Grid

Alert as required.

14   Weather Monitoring: Weather Insert

14.1 Event:	
Event 1:	Insert WX_GRIDS

14.2 Condition:	Comment
WX matches monitored grid	

14.3   Action:

Evaluate Weather Rules for Mission Grid

Alert as required.

15   NOTAM Monitoring: Event Insert

15.1 Event:	
Event 1:	Insert LOC_MSN_EVNT

15.2 Condition:	Comment
LOC_MSN_EVNT_TYPE_CD ='L'	

15.3   Action:

@notam.sql

Alert as required.

16   NOTAM Monitoring: Continuous Montoring

16.1 Event:	**Periodic(Event 1, 24 hrs, EVENT_COMPLETION_TIME)**
Event 1:	Insert LOC_MSN_EVNT

16.2 Condition:	Comment
LOC_MSN_EVNT_TYPE_CD = 'L'	

16.3 Action:

@notam.sql

Alert as required.

Notify Action

1   Alert Notification

The user will select the method for alert notification along with a prioritization scheme. For example, if the first choice is "Workstation Alert" and the user is not logged on, then the alert should be delivered via a secondary mechanism e.g. pager, email.

Registry for alert notification will require identification by either role or user name.

1.1   Workstation Alert

Delivery of URL via either Jabber mechanism or our "roll-your-own" mechanism.

1.2   Email Alert

Email alert should include URL to access full alert and at least the top level textual alert. Subject should include priority of the alert.

1.3   Pager Alert

Identical to email alert with the exception that the alert may not be a URL and must be restricted to character limit of pager.

Appendix F: Information Dominant Decision Environment Demonstration Instructions

1   Login to delta1

Username	jkraska
Password	PlebeZ66

At the command prompt execute the following scripts:

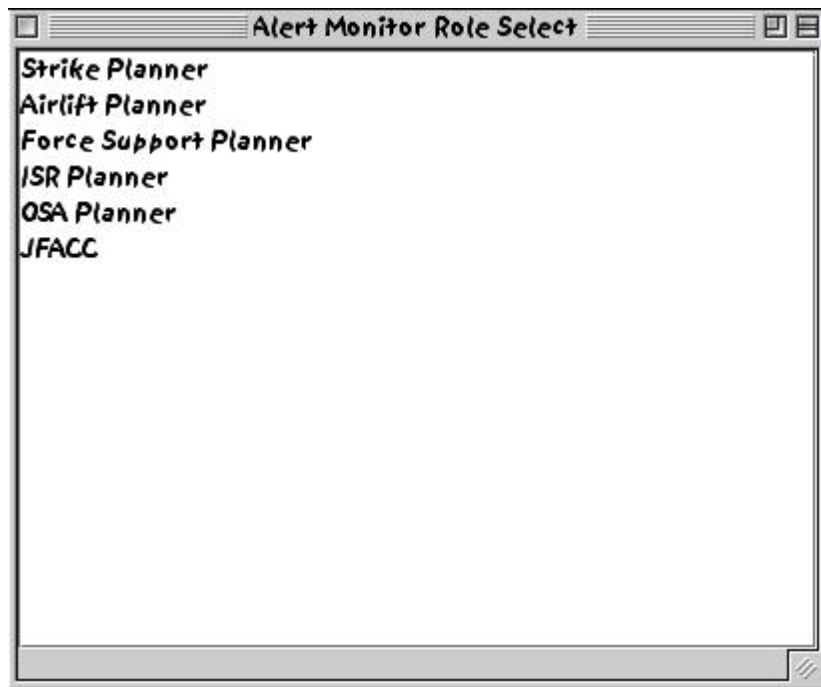## 1.1   setup-demo

1.2   update_data

1.3   run_demo

2   On the client machine (e.g. windoze laptop)

2.1   Run caccGui by double clicking the icon

You should see the following in a java console window.

2.2 Select a role for receiving alerts

Role	Types of Alerts
Strike Planner	Mission Weather, Mission Cancellations and delays, Time Critical Targets
Airlift Planner	NOTAMs for scheduled Airlift Missions
JFACC	Sustainability analysis for Air Campaign Plans

You should see several alerts popup in a small window on your screen. The priority of the alerts is indicated by the color. Click on an alert with your pointing device (e.g. mouse) and the alert information will be displayed.

Type of Alert	Information Displayed
Mission Weather	List of affected Missions, the type of weather, and the type of operations.
Mission Cancellation or Delay	List of affected Missions
Time Critical Targets	Map showing location of TCT and enemy defensive units currently known. Icons showing location of Air Alert Aircraft currently available. Click on the aircraft icon to see a list of available aircraft and their weapons load. Icons showing the location of Ground Alert Aircraft currently available. Click on the base icon to see a list of available aircraft and their weapons load.
NOTAM	Your browser will be started up and pointed to a URL listing the NOTAMS for the selected base.
Sustainability	Your browser will be started up and pointed to a URL containing a complete ICIS analysis of the sustainability of the campaign plan. The ICIS URL is currently only reachable from within the BBN firewall.

Windows Installation Instructions

1.1   Download and install jdk 1.2.2

Download the file jdk1.2.2Win.exe and run the install procedures. Recommend that you install jdk1.2.2 on the C: drive at the root level. Note the directory path where you install jdk.

1.2   Modify your autoexec.bat

Your path variable must include the directory path for jdk1.2.2/bin. The figure below shows a typical autoexec.bat file with the added information highlighted in GREEN. If there is no PATH variable in the autoexec.bat then you must add one.

```
SET BLASTER=A220 I5 D1 T6
PROMPT $p$g
PATH=C:\WINDOWS;C:\WINDOWS\COMMAND;C:\CNTX;C:\JDK1.2.2\bin
rem - By Windows Setup - C:\WINDOWS\COMMAND\MSCDEX.EXE /S /D:IDECD001
/L:E

C:\CNTX\SYSINI.EXE
```

1.3   Reboot your machine

This initializes the machine with the new information from the autoexec.bat

1.4   Download the AlertMonitor client software caccAlertMonitor.jar

1.5   Open a DOS Prompt window

1.6   Execute jar xvf caccAlertMonitor.jar.

This will install the Alert Monitor client software in the current directory. Note the directory path.

1.7   Configure the alert mechanism

Open the file caccAlertMonitor\rmiVersion\cacc.properties in a text editor and set the four variables to the appropriate paths. The file path separator for windows is a backslash \. On windoze machines the backslash must be preceeded by a backslash so that the path separator will be properly recognized.

HostName is the alert server. You must ensure that your machine has network access and access permissions to the alert server.

Browser: You may use any standard browser. You must indicate the full path to the browser executable.

Grammar: Modify this to reflect the directory path where you installed caccAlertMonitor

Archive: This is the directory where alert data will be stored. It must have a file separator at the end. On Windoze machines that means two backslashes.

```
# This is the properties file for the cacc alert monitor

# Where is the alert server running?
# can use dns or ip address
HostName = 128.132.33.131
# Which browser to view alert data, and where is it (spaces ok in path name)
Browser = \\C:\\Program Files\\Netscape\\Netscape.exe
# Which file is being used to retrieve Mode list and where is it
Grammar = \\C:\\caccAlertMonitor\\rmiVersion\\grammar.igr
# Where is map and table data being stored?
# Must have trailing \\
Archive = C:\\ALERTS\\
```

Appendix G: Global Event Detection Enhancements

ENHANCEMENTS TO THE GLOBAL EVENT DETECTOR
TO IMPROVE
FUNCTIONALITY AND PERFORMANCE

By

GAURI SUKHATANKAR

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

1999

To my family

ACKNOWLEDGMENTS

TABLE OF CONTENTS

LIST OF FIGURES

ENHANCEMENTS TO THE GLOBAL EVENT DETECTION SERVER

TO IMPROVE FUNCTIONALITY AND PERFORMANCE

By

Gauri Sukhatankar

May 1999

Chairman: Sharma Chakravarthy

Major Department: Computer and Information Science and Engineering

Sentinel is an active object oriented database management system (DBMS) that monitors conditions associated with events and allows the specification of event-based rules. Events include method, temporal and external events. When an event is triggered, the condition associated with that event is evaluated and if it evaluates to true the action is executed.

The global event detection server (GED) has been designed to facilitate Sentinel applications to define rules on events occurring outside of their address space. The GED has a communication architecture consisting of sockets and remote procedure calls (RPCs) for passing external events across applications. The GED monitors external events using the producer consumer paradigm. In the distributed application environment it is important to accommodate various kinds of failures: consumer failure, producer failure as well as GED failure. To accommodate for failures,

the GED persists events and supports recovery. To accommodate for very high rates of event generation by the producer clients, buffer management was added to the GED.

Although the GED monitors external events and is recoverable, currently it cannot support multiple clients efficiently. Its performance needs to be improved in order to accommodate multiple producers and consumers exchanging a large number of events at any given time. Currently the GED passively passes events across applications without any computation on them. To facilitate "intelligent" forwarding and "filtering" the GED needs to be able to check some conditions before passing events. An expressive GED server will be one that has the capability to support Event Condition Action (ECA) rules on global events. The focus of this thesis is to make the GED scalable and improve its performance, and to extend the GED functionality to support ECA rules on global events.

CHAPTER 1
INTRODUCTION

An active database monitors the database state and reacts spontaneously when predefined events occur. Functionally, an active DBMS allows specification of event-based ECA rules and monitors the conditions associated with events. Events include domain specific events such as insert, update, delete for relational databases, method invocations for object oriented databases and temporal and external events. An object oriented database system supports user defined data types and provides powerful capabilities to model and persist complex objects. A number of emerging applications, such as computer aided manufacturing, power distribution network, automated office workflow control need to continually monitor their database state and quickly respond with proper actions to certain events. When an event occurs, the condition function is evaluated and if it evaluates to true, the action is executed. Thus, an active database supports applications with ECA rules. This active capability is useful in a single system. For applications that access more than one database we need to extend the active capability to work in a distributed environment.

In order to support active capability within applications, the local event detection server was designed. The LED allowed an application to define rules on local events. The GED was designed to extend the event specification capability so that an application can specify rules on events external to its address space. In order to support global (external) event specification and detection in a distributed environment, global event definitions were added to SNOOP (an event specification language of Sentinel). The LED was also extended to support for communication with the GED and was called ELED (extended LED). The Global event detector (GED) was implemented to detect events that span multiple applications. Client applications connect to the GED server and register events with it to use its capability of global event notification. When external events are sent to the GED, it sends event notifications to consumer clients over a socket and the clients make remote procedure calls to send or receive events from the GED.

Although the GED monitored events in distributed applications, reliability issues had not been addressed in the initial design. In order to address robustness and recovery issues, reliable event detection and prorogation was needed. Hence, a recoverable GED was implemented that gets to a

consistent state following various types of client failures and can also continue to provide services in a normal fashion when the GED itself recovers from system failures. The GED uses write-ahead logging for persistence as well as recovery. Buffer management is used to support flexible allocation and use of memory by the GED, as unlimited memory availability for storing events is not realistic. Events are stored in an event file (one per consumer) and in a buffer (whose size is specified in the GED configuration file) from where they are received by a consumer.

1.1 Motivation

The current work on the Global Event Detector supports event monitoring and recovery in a distributed application environment. However, it does not address the issues of scalability. When multiple clients connect to the GED and exchange a large number of events, performance of the GED must not deteriorate. Since the current implementation uses synchronous RPCs, a client process making a request to the server blocks will wait until a reply is returned. If multiple clients are making requests to the GED at the same time then a lengthy request by one client will cause the other clients to wait. Moreover, response time for clients will increase as more and more clients connect to the GED. To reduce these delays the GED should be able to handle multiple client requests concurrently. Hence the GED server needs to be a multitasking server. However, multitasking will involve concurrent access to shared data structures, which may cause race conditions. Hence these shared data structures also need to be protected.

The current implementation of write-ahead logging and buffer management has not been done in an efficient way. Each log record contains an event occurrence and its parameters and has a unique LSN, which is its log sequence number. Reading of a specific log record, given its LSN, involves reading the entire log file from the start to the position of the record LSN. This takes considerable I/O time, especially when the log file is large. Knowing offset of BLSN (LSN of last event in the buffer) will allow one to read an event without sequential read of the file. Storing the log in binary format instead of ASCII will make reads and writes faster.

In the current implementation the GED is just a passive mediator of events across different applications. In a typical situation the GED needs to have the capability to do filtering, delaying, or useful computation on the data associated with received events before sending events to the consumer. This means that the GED needs the capability of defining rules where useful actions can be taken under certain conditions on events arriving at the GED. A rule editor is currently being used in Sentinel to allow the user to specify rules on local events within each application. To support specification of rules on global events, the existing rule editor interface needs to be

extended. A rule editor server also needs to be designed that will do operations such as compilation, persistence and retrieval on the rules specified through the interface. The GED must support dynamic loading of these persisted rules at runtime so that rules will be fired when global events are notified to the GED. The focus of this thesis is to make the GED a multithreaded server to handle client requests concurrently, to provide efficient logging and buffer management and to extend the GED functionality to support ECA rules on global events.

The remainder of this thesis is organized as follows: Chapter 2 presents an overview of the event specification language and current support for rules in Sentinel. Chapter 3 discusses some design issues, analyzes their advantages and disadvantages, and describes how they are applied in the multithreading of the GED. In Chapter 4, we discuss more implementation details of the GED. Chapter 5 gives the performance measurements on the GED. Chapter 6 discusses design issues for supporting rules in the GED and Chapter 7 discusses implementation of rule support. Chapter 8 gives a conclusion along with discussion of the limitations and future work.

CHAPTER 2
OVERVIEW OF SENTINEL

Sentinel is an active OODBMS with an integrated approach. It supports primitive event detection and nested transactions as part of its kernel. In addition, it supports composite event detection and rule management as separate modules. Snoop [1] is an event specification language used in Sentinel for specifying ECA rules. It defines event and rule specification, supports event operators and parameter contexts. SPP is the preprocessor for SNOOP.

2.1 Types of Events

Four types of events can be identified in a distributed system:

<u>Local primitive event</u>. Any method of any object class is a potential primitive event. A local event is one which is predefined in the application. Primitive events include database events and temporal events. **Database events** correspond to database operations such as insert, delete or a method invocation on an object. Events are further refined into *primitive* events by using event modifiers, begin or end. For example, every instance of a *Stock* class may have a method *Insert*. Then *Stock* can potentially invoke its *Insert* method and produce an event, such as *begin-of Insert*. **Temporal events** include absolute or relative temporal events. An absolute temporal event is specified as an absolute value of time such as a time string using the format <(hh/mm/ss)mm/dd/yy>. A relative temporal event corresponds to a unique point on the time line but in this case both the reference point and offset are explicitly specified. A relative temporal event is given as follows:

Primitive_event ::= event *event_modifier* method_signature

<u>Local Composite Event.</u> Local composite events are formed by applying a set of operators to local primitive events and local composite events. If E1 and E2 are either primitive or composite events then a local composite event is given as follows:

Composite_event ::= E1 operator E2

111

Global Primitive Event. Global primitive events are events that are either local primitive or local composite in one application and are referenced/used by another application. Since a global primitive event occurs outside of the application that uses it, there must be some way to define the event as well as to detect and transmit the occurrence. *App_name, Remote_event_name* and *Host_name* are attributes that solve this problem. *App_name* is the name of the remote application or its ID. *Remote_event_name* is the name of the event defined locally within the remote application. *Host_name* is the name of the host on which the remote application is running.

Global_primitive_event ::= *Remote_event_name::Host_name__App_name*

Example: E1 ::= produce::rain__prod

Global Composite event**.** If an event is composed of one or more events where at least one of them is a global event then it is called global composite event.

Example: compE ::= E1 operator E2

      (at least one of E1/E2 is a global event)

2.2 Parameter Contexts

Snoop supports the notion of parameter contexts [2] to capture application semantics for computing parameters or consuming event occurrences (of composite events). These contexts are precisely defined using the notion of initiator and terminator events. An initiator of a composite event is a constituent event that can start the detection of the composite event whereas a terminator is a constituent event that can detect the occurrence of the composite event. A composite event may be comprised of several primitive events that can be initiators and terminators of another composite events. When the occurrences of several primitive events constitute a composite event, different possibilities for detecting composite events exist. By carefully analyzing several classes of applications, four parameter contexts are proposed in Snoop:

Recent: In this context, only the most recent occurrence of an initiator for any event is used. All other occurrences of a constituent event will not be used in detecting another composite events and they will be deleted when the event started by the initiator occurs. The initiator of the event will continue to initiate another event until another initiator occurs.

Continuous: Each occurrence of the initiator of an event continuously initiates the event. A terminator may terminate one or more occurrences of the same event.

Chronicle: The initiator and terminator pair of a composite event is unique and deleted after the event occurred. They are paired based on chronological basis.

Cumulative: In this context, for each constituent event, all occurrences of the event are accumulated until the composite event is detected. In other words, the parameters of a composite event include the parameters of all the occurrences of each constituent event. All the occurrences of each constituent event are flushed whenever its associated composite event is detected. Detailed discussion of the parameter contexts is given in [3].

2.3 Event Operators

Figure 1 shows the types of events in Sentinel. The following is the summary of Snoop operators with brief explanations:

AND: Conjunction of two events, namely E1 and E2. The order of occurrence of E1 and E2 is irrelevant. example: E_AND ::= E1 ^ E2

OR: Disjunction of two events, namely E1 and E2, occurs when either E1 or E2 occurs. Example: E_OR ::= E1 || E2.

SEQ: Sequence of two events, namely E1 and E2. Occurs when E2 occurs after the occurrence of E1. Example: E_SEQ ::= E1>>E2.

NOT: Negation operator detects non-occurrence of an event, namely E2, in the closed interval formed by two events, E1 and E3.

Example: E_NOT ::= ¬ E2[E1,E3].

A: Aperiodic event is detected for every occurrence of E2 during the half-open interval formed by E1 and E3. Example: E_A ::= A(E1,E2,E3)

A*: Aperiodic closure event is a cumulative variant of the A operator. It is detected when E3 occurs provided E1 has already occurred. The occurrences of E2 are accumulated during the half-open interval formed by E1 and E3.

Example: E_A_STAR ::= A*(E1,E2,E3)

P: Periodic event is detected for every time period specified by E2 during the half-open interval (E1,E3]. E2 is a time specification.

Example: E_P ::= P(E1,E2,E3).

8. P*: Periodic closure event is a cumulative variant of P operator. A P* event, where E2 is a relative temporal event, is detected only once when E3 occurs provided the E1 has already occurred.   Example: E_P_STAR::= P*(E1,E2,E3)

Figure 1: Types of Events in Sentinel

Event Classification

Event Classification Event Classification Events

Local Events   Global Events

Primitive   Composite   Primitive   Composite

Method   Temporal

Absolute   Relative

Event Operators

Event Operators

AND   OR   SEQUENCE   NOT   PLUS   ANY   PERIODIC   APERIODIC

P*   P   A*   A
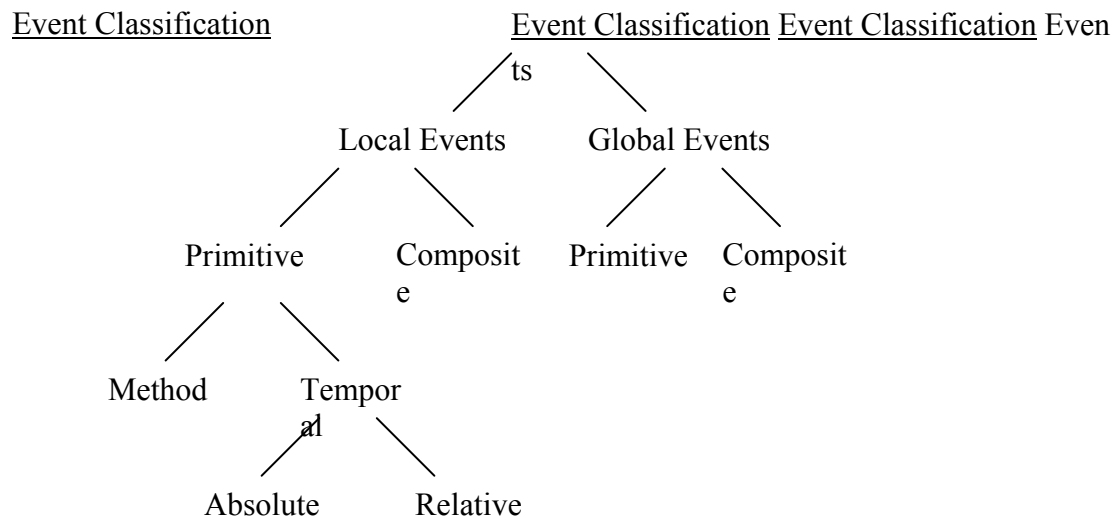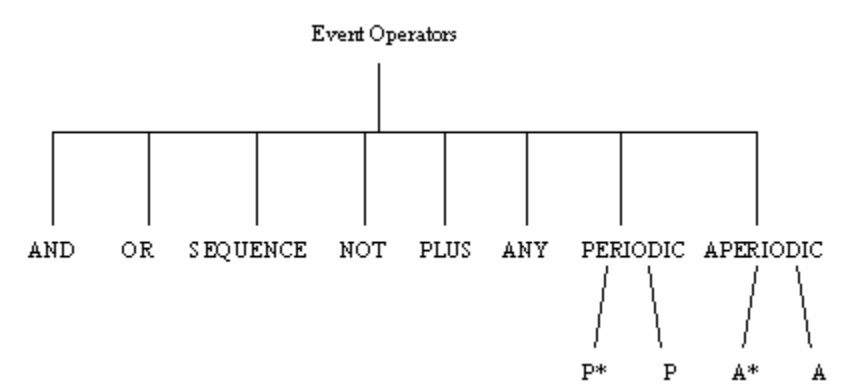
2.4 Summary of Event Detectors

In an object oriented DBMS, database events correspond to the execution of methods of a class. Therefore, there must be a mechanism to trap the invocation of (or return from) a method

when an event is signaled. In a centralized system the Local Event Detector (LED) [2] is used for detecting local primitive events and composite events within applications. In a distributed computing system, events across distributed applications need to be monitored. To accommodate global event detection, some extensions were made to the LED. The new LED is called ELED. Global Event Detector (GED) [4] is responsible for monitoring events from different applications in a distributed environment. It recognizes the occurrence of events, collects and records their parameters, and passes them to the interested applications where the rule managers trigger the action of ECA rules.

2.4.1 Local Event Detector

The purpose of the local event detector (LED) is to detect the occurrence of local primitive and local composite events within an application. The LED is implemented as a class and we have a single instance of this class per application. It is linked with an application for detecting local events. The REACTIVE class is a class that contains procedures for dealing with the event and rule specification. Every method of a REACTIVE class is a potential event. The LED is an EVNT_LIST, which is a linked list of EVNT_NODE. Each EVNT_NODE corresponds to a unique REACTIVE class. EVNT_NODE has a *begin_of* event list and *end_of* event list. *begin_of* and *end_of* correspond to primitive events that should be raised at the beginning or at the end of the method, respectively.
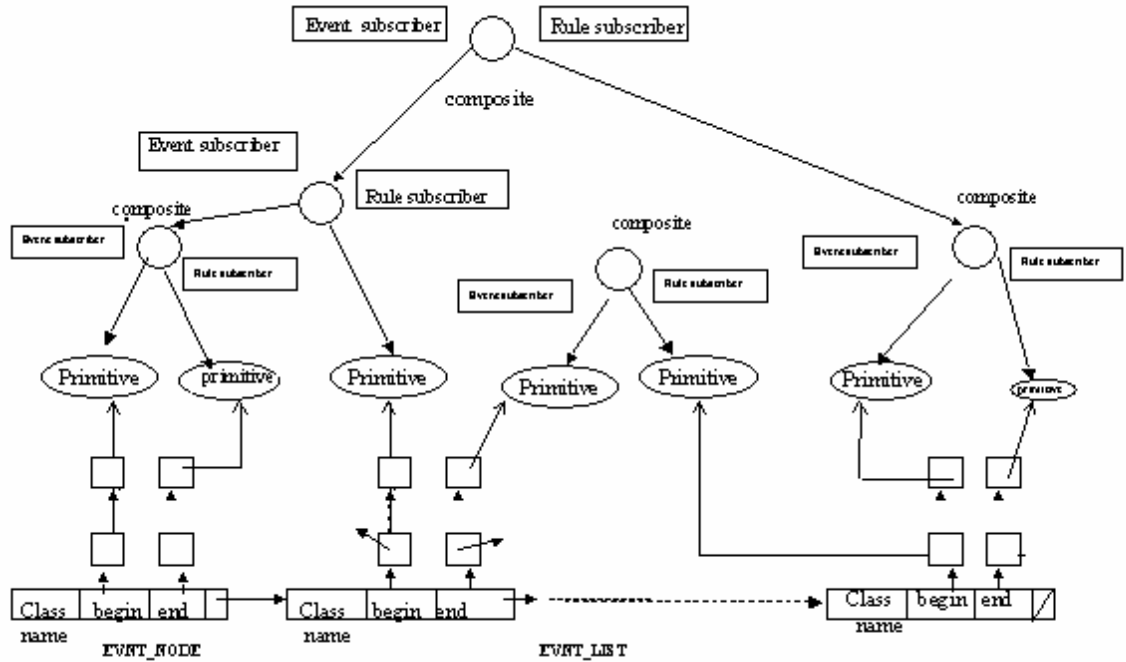
Figure 2: LED Data Structure

An event graph is used for event detection. The data structure of the local event graph has been shown in figure 2. Each node of the two events lists points to a primitive event that is a leaf node of the event graph. The leaf nodes of the event tree graph correspond to primitive events from which the composite events are constructed. Each node of the event graph has an event subscriber and a rule subscriber that records the related composite events and rules. Whenever a primitive event is raised, it will notify its subscribers which are its parent nodes. The parent nodes (composite events) will maintain the occurrence of its constituent events as a part of its parameter lists, which are stored separately for each context relevant to the node. If the composite event occurs by the current notification, it is detected and notified to its subscribers. The parameter list is recomputed to include the new occurrences. For details of the LED, refer to [2].

In order to support rules on global events in the LED, the LED interface was enhanced to communicate with the GED. This extended version is called ELED. In addition to detecting local events, the ELED will have to send an event notification to the GED server when an event is raised that is needed by other remote sites. Moreover, the ELED will detect a global event only when it receives an event notification from the GED server. To accommodate global events, a

REMOTE class has been added to the class hierarchy in the LED. Similar to the LED, the ELED is an instance of the EVNT_LIST class that records information of all the event instances. Each node of the EVNT_LIST is related to a unique application and contains all the global event instances that are detected outside of this application. Each node contains a list (ELIST) of REMOTE nodes that become the leaves of the event graph. Whenever a global event is detected outside of the application, the GED interface will receive the event notification along with application ID and event parameter list from the server and further notify the ELED. ELED then determines the specific EVNT_LIST node according to the application ID and propagates the event notification to its corresponding REMOTE event instance. According to its event subscribers and rule subscribers, a notified REMOTE event instance will further notify related composite events, that is its parent nodes.

2.4.2 Global Event Detector

The global event detector detects events that span several applications in a distributed application environment. Its purpose is to allow an application to detect events occurring not only at a local site, but also at other remote sites. It recognizes the occurrence of events, collects and records their parameters, and passes it to application rule managers where an ECA rule will be triggered.

Since each application has its own local event detector, a global event detector is responsible for detecting events that are defined at a remote site. Therefore, the global event detector (GED) adopts the client/server model and must be able to communicate with local event detectors at remote sites through RPC and socket-based communication. Figure 3 shows the GED communication architecture. Detailed discussion of the GED architecture alternatives is in [4].

First, a client process makes a socket connection to register with the GED server, and the server will record the socket address of this client and events that need to be detected by the GED if this client is a consumer. Event name list (*cname_l*) will be sent to the corresponding producer who generates the requested event, and the *GED_forward_flag* will be set to 1 with corresponding event in *cname_l*. Then, whenever the LED detects an event, it checks its *GED_forward_flag* to see if this event needs to be sent to the GED server. Global primitive events are first detected by the local event detectors at their corresponding remote sites. Then, event notifications are sent to the GED server. When an event is notified to the GED server, it sends a message to each consumer on a socket. After the consumer has received this message, it makes a remote procedure call to the server and pulls the event (with its parameter list) from the server. Finally, the consumer traverses its ELED graph to propagate the global event.
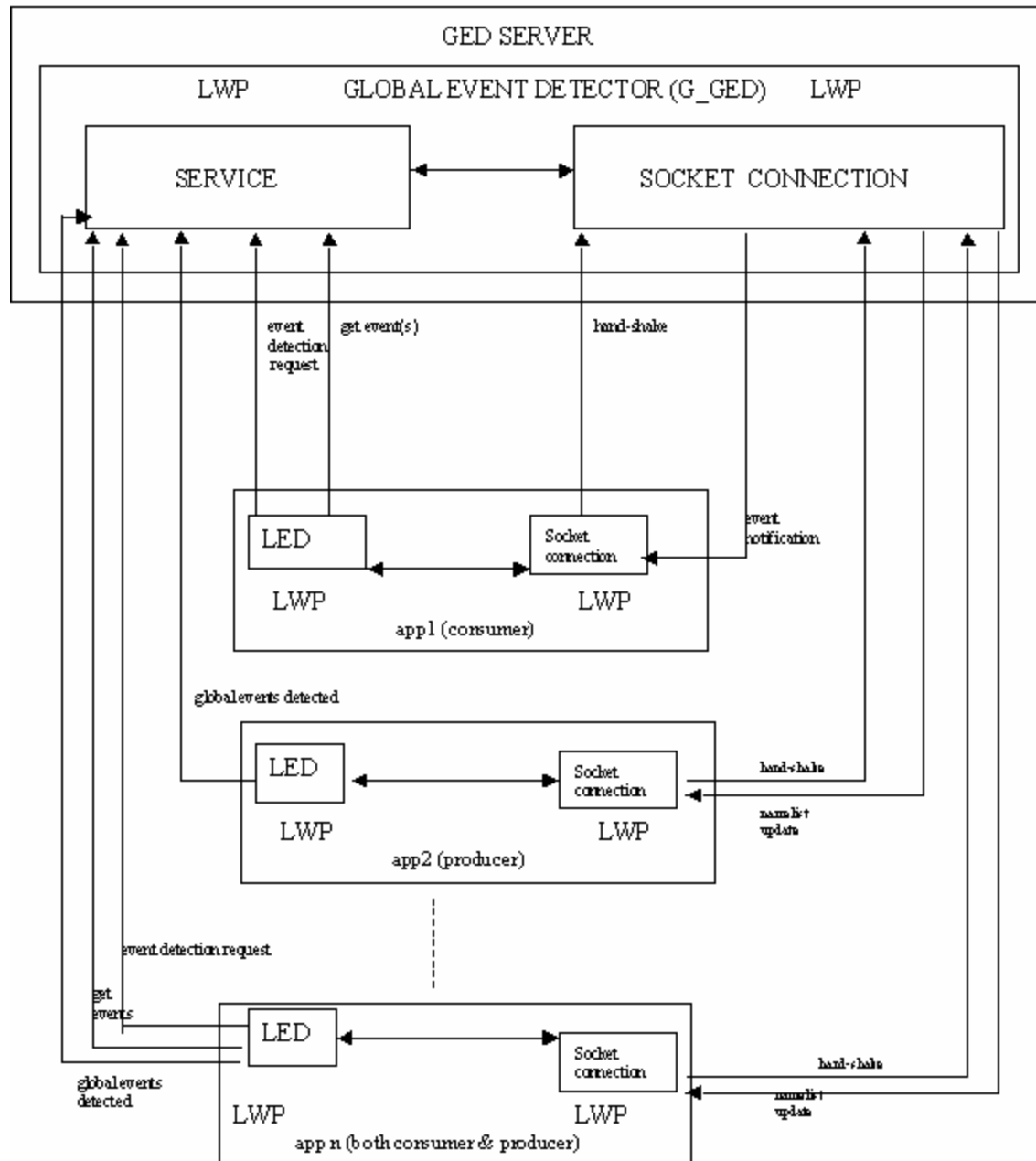
Figure 3: GED Communication Architecture

2.4.3 Global Event Graph

A PRIMITIVE class in the LED specifies primitive event objects; however, it is not appropriate to use this terminology in the GED since global primitive events denotes external events that are detected outside of the local application. Therefore, the GLOBAL class was introduced instead. GLOBAL class stands for the global primitive event objects. There are three attributes in GLOBAL class: *send_sname, send_ename, and event_no*. *send_sname* indicates the (consumer) application ID (machine name__application name) that is to be notified by the server after this event is raised. *send_ename* is the name of this event that application *send_sname* uses. It has the same value of *ename* attribute of a REMOTE class instance which is related to this global primitive event in application *send_sname*. *Event_no* denotes the instance number of the occurrence of this event.

Whenever a client registers with the GED server, it must send some information to the GED server to build the global event graph (G_GED) if global events are defined in the client. The information that a client sends is obtained from the global event specification file that is generated by the *sPP*. Refer to [4] for more information on the global event specification file. Similar to the LED, the G_GED is also an instance of EVNT_LIST. However, the NOTIFIABLE class that each ELIST points to is a GLOBAL class instead of PRIMITIVE or a REMOTE. So, when a global event is notified to the GED server, it traverses the G_GED and further notifies related composite events (parent nodes), and computes its parameter list. Figure 4 shows the data structure of the GED global event graph.

Global events are detected on the server using the event graph. An event tree is created for each composite event and these trees are merged to form an event graph for detecting a set of composite events. This will avoid the detection of common sub-events multiple times thereby reducing storage requirements. For each node in the event graph, there is an event subscriber linked containing all the composite events that use this event as its constituent one. An event node has a pointer to its subscriber which becomes its parent node. Whenever a global primitive event is detected, it will propagate the event notification to its subscribers, that is its parent nodes. Event occurrences flow upwards as in a data-flow computation. The parent nodes maintain the occurrences of its constituent events along with their parameter lists which are stored for each context set to the node. If the composite event occurs by the last notification, it is detected and further propagates to its subscribers. Each time an event is raised, it will send this event to a specific application that had subscribed for the event.

1.1 2.5 Support for Rules in Sentinel

A sentinel application has the capability of defining Event Condition Action rules. The following paragraphs give a brief summary of the rule support in sentinel applications.

In the context of Sentinel, a rule consists of an event, a condition function of boolean type, action function of void type, and a few attributes. Once an event is detected by the system, the associated condition function is evaluated and the associated action function will be executed based on the result of the evaluation. Sentinel supports nested rules. When a rule's action raises an event that triggers rules there is a nested execution of rules.
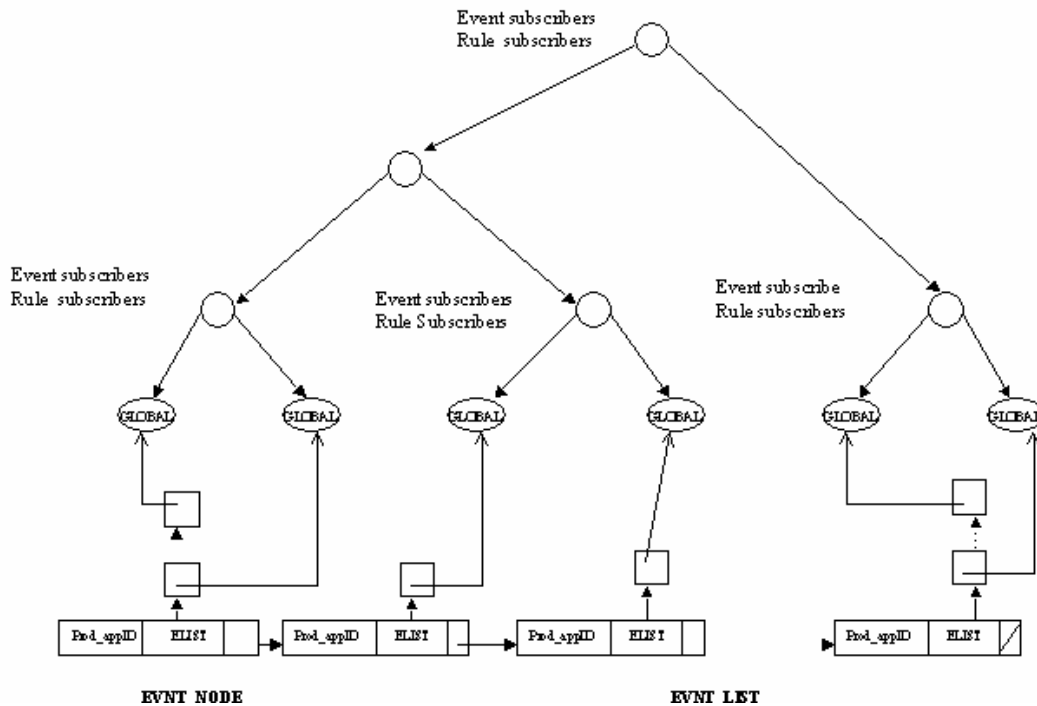
Figure 4: Global Event Graph

Detailed discussion of rule semantics is in [5]. As part of the rule semantics, it is necessary to know when to execute condition and action functions after the associated event has been raised. This is an issue about the *coupling* between the event and the condition-action pair. Currently, *immediate* and *deferred* coupling modes are supported between event and condition-action pair.

Sentinel supports multiple rules. An event can trigger several rules. Therefore, it is necessary to support rule execution mode that supports concurrent and prioritized serial execution of rules. Sentinel uses *Priority* classes for specifying rule priority. An arbitrary number of priority classes

can be defined. A rule is assigned to a priority class by indicating its number or the name of the class. Sentinel provides a global conflict resolution mechanism among the priority classes and concurrent execution of rules that belong to the same priority class.

Sentinel also introduces two types of rule *trigger* modes for specifying the time from which event occurrences are to be considered for the rule. The two trigger modes are *now* (start detecting all constituent events starting from this time instant) and *previous* (all component events since the event was detected last are acceptable). *Now* is the default trigger mode. Following is an example of a rule defined within an application on a global primitive event g1:

event g1 = STOCK_e1::manatee__app1;

    rule gr1[g1, cond1, test_act1, RECENT];

The rules supported by Sentinel are specified in the classes (class-level) and the code (instance-level) of an application. They are referred to as internal (static) rules. Internal rules are specified in the application and processed. The application source code is processed by the Sentinel preprocessor (sPP), whose output is then processed by the Open OODB preprocessor and finally compiled by a C++ compiler into an executable. Rules on events within an application can also be specified externally by the user by using a rule editor. The condition and action functions specified by the user through the rule editor are compiled and rule information is persisted into the database using OQL queries [6]. Function names of the condition and action are persisted with the rule information into the database and the compiled function object code is made into DLLs. At application runtime, the *load_dyn_rules* routine (available as part of sentinel code) is called which uses the condition and action names to retrieve the function pointers by using the dynamic loader utilities, *dlopen* and *dlsym*. In this way externally defined rules can be loaded into applications at runtime. Details of the rule editing and loading are in [7]. Whenever an event is detected within an application the condition function will be executed and if it evaluates to true the action function for the rule will be executed.

CHAPTER 3
DESIGN ISSUES FOR MULTITHREADING

This chapter discusses our approach to making the GED multithreaded. In addition to multithreading, several performance related improvements have been incorporated into the GED code.

3.1 Design Goals

The GED server was designed to monitor events in a distributed application environment where the client applications are producers and/or consumers of events. Consumer clients make RPC calls to register events with the server. Similarly producer clients make RPC calls to send instances of events to the GED. Consumer clients also make RPC calls to receive event occurrences when the GED notifies them. An RPC call is synchronous and blocking [8]. The mechanism of synchronous RPC is shown in figure 5. This means that a client making an RPC call to the server is blocked till the call returns. In the current implementation of the GED, the RPC request service procedure is in the main thread of the server process. Therefore, even if two or more clients are making procedure calls at the same time they will be handled serially by the server, and a client may have to wait for service till the server finishes servicing the previous client request. This wait will be significant when the server is handling several clients and the events being delivered are large. In order to make the GED scalable it should be able to handle multiple client requests concurrently. Hence the first design goal is to have a multitasking server, as shown in figure 5. Multitasking can be achieved either by multithreading or by forking child processes, as explained in [9]. In case of multithreading each RPC request will be serviced in a separate thread. When service procedures are being executed concurrently, two or more threads may be accessing the same data structures at a given time. To prevent race conditions, appropriate synchronization mechanisms must be provided for protection of data structures. However, locking of data structures must not be so coarse grained that it will effectively serialize their access. Hence synchronization mechanisms must be carefully chosen to a fine granularity of locking and to maximize concurrency. The GED server also is recoverable, which means that the server as well as clients can recover from crash in a way that is transparent to the other

applications. This is achieved by using write-ahead log mechanism. In order to speed up the GED, a better format and algorithm is required for writing and reading the consumer log files. File access will also be quicker if log records are written in a binary format instead of ASCII as is currently being done.

1.1   3.2 Multithreading the Server

The GED server uses RPC and socket protocols in its communication interface. In order to listen for client requests when the server starts running, the server process first registers the program, procedures and version numbers with *registerrpc* command. The port mapper then advertises the availability of the RPC address so that interested clients can open a channel with the server. The server then goes to sleep while the *svc_run* call listens to the other end of a socket for a client request to come along.
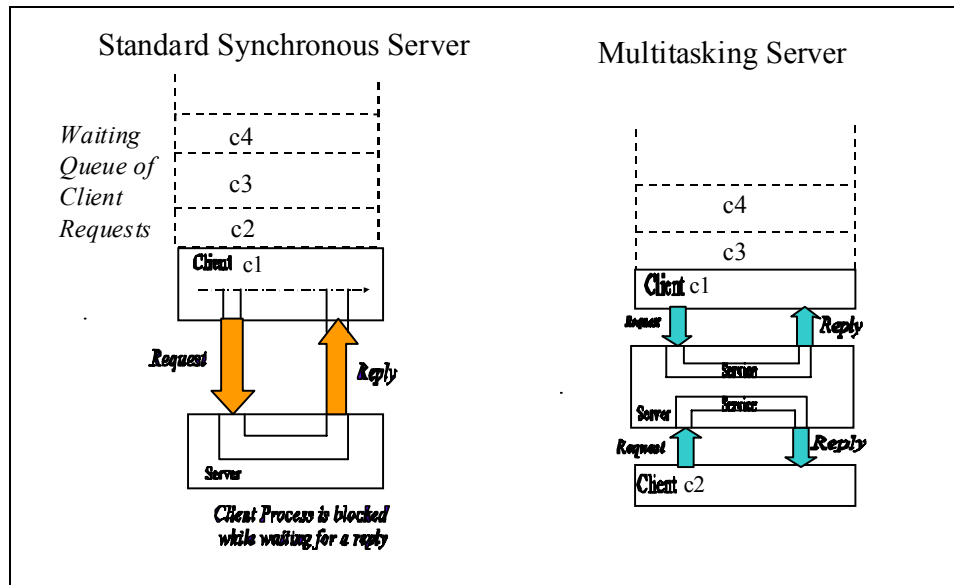
Figure 5: Synchronous RPC Server vs Multitasking Server

Based on the client request (argument), it executes one of the procedures mentioned in the *registerrpc* call and returns the reply (result) to the client. The *svc_run* routine is the heart of the server. Typically, it loops indefinitely, checking a set of socket descriptors. When it gets a service request, it switches to the associated procedure on examining the type of request. Once the procedure is executed, it loops back and waits for additional requests. Details of *svc_run* are in [8]. In order to make the server **scalable** it must execute each procedure in a separate process or thread, while the main thread (or process) listens for additional requests. Multi-tasking the server is useful mainly when there are multiple client requests that take widely different times to

123

process. Even when the server is stuck in the middle of a request which is a long processing task, the server should be pre-empted by other brief or higher priority requests.

Multitasking can be achieved by either forking a process or allocating a thread to handle each request. In order to handle the request in a separate process, a child process can be forked during dispatching of the request. Another alternative would be to start a child process for each service procedure when the server is first started. In order to use multithreading instead of child processes, the *svc_run* routine must create a pool of threads where an idle thread will be allocated to handle the next request. The *rpc_control* utility function provided by the RPC library provides an option of starting the server in the AUTO_MT mode wherein the procedure dispatch routine in *svc_run* allocates a thread to handle each service request. Figure 6 shows the details of multithreading.

A thread is a single flow of control within a process. Threads share a single address space. Each thread shares the resources of the parent process. Although multitasking can also be achieved by creation of child processes, multiple threads of execution provide much higher performance as compared to full-blown forking [10]. First, since threads share global variables, memory sharing is not an issue with threads. Second, while context switching among threads, only pointer to the thread's stack and registers needs to be **s**aved. For process context switches, all registers, stack, data, program counter as well as several runtime state parameters of process need to be saved. Hence context switching for threads is a lot cheaper than for processes. Third, when processes synchronize, they usually have to issue a system call, a relatively expensive operation that involves trapping into the kernel. But thread synchronization is usually handled by the runtime thread library, and is less expensive as it does not require a trap to the kernel. For the above reasons multithreading was seen to be the better of the two alternatives to achieve multitasking.
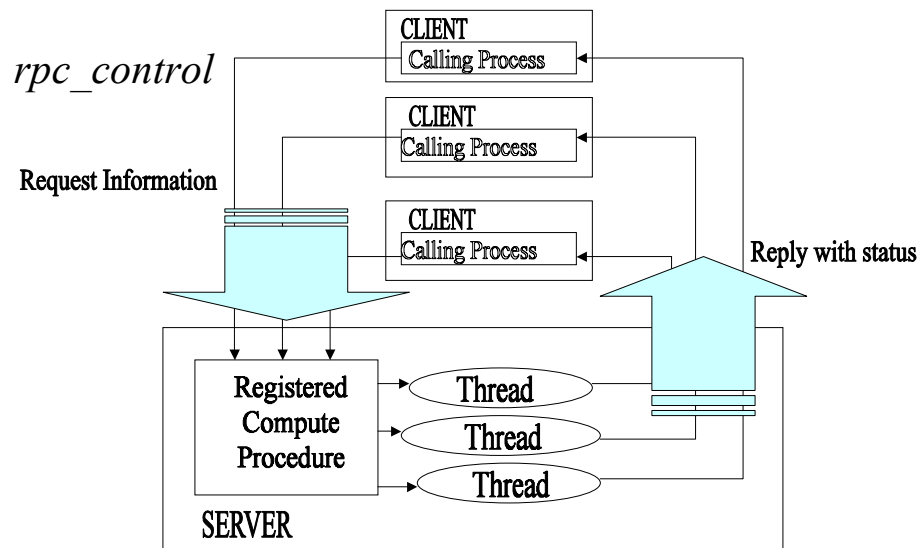
Figure 6: Details of Thread Handling

3.3 Synchronization Issues

The GED code is made up of several data structures that will be shared and hence may be concurrently accessed by threads. Following is the list of shared data structures:

Client address list (*client_addr_list):* list of client names and socket addresses.

Consumer event list (*event_para_list*): list of event consumer clients.

Global event graph (G_GED): graph of events with parameter lists to be propagated.

Producer list (*site_evnt_list*): list of producers.

Consumer list (*event_para_list*): list of consumers.

Event counter (*event_counter*): counter of total events received by the GED, used for assigning LSNs.

Event Log file (*consumer_name.log*): There is one event log file per consumer, that has one log record for each event occurrence to be received by the consumer.

8. Global event file (*GED_spec.log):* log of events in the GED.

9.   Client Address file (*client_addr.log):* Client Ids and respective socket Ids.

Race Conditions. When the result of two or more threads performing an operation depends on unpredictable timing factors, there is race condition. Example of a race condition: Thread A is in the process of deleting a consumer node at position 7 from the *consumer_addr_list.* Thread B is

125

traversing the *consumer_addr_list* to get the socket address of a consumer node at position 13 to which it wants to send a message. Thread B could be looking at node 7 when the list manipulation is occurring. Thread B will decide that node isn't the desired node and go to the next position in the list. However, since thread A has disconnected this node from the list the next position could be NULL. The result of what thread B reads will hence depend on the timing factor and has been compromised by the race condition. Hence the access of the *consumer_addr_list* and several such shared data structures must be guarded for mutual exclusion. This can be attained using synchronization mechanisms or locks. There are several types of locks and the right choice must be made.

3.3.1 Types of Locks

Mutex lock is a synchronization primitive that allows multiple threads to synchronize access to shared data by providing mutual exclusion. The mutex lock has only 2 states: locked and unlocked. Once a thread has acquired the mutex lock on a data structure other threads attempting to lock the structure will be blocked until it is unlocked. Since mutex allows only one thread to access any data at a given time, it is the most restrictive type of access control. For example, when a mutex is used to synchronize access to a list, the mutex will control the entire list. While the list is being accessed by one thread it is unavailable to all other threads. If most accesses are reads and writes of the existing nodes as opposed to insertions and removes, then a more efficient approach will be to allow nodes to be individually locked.

Read-write lock is another synchronization primitive that was designed specifically for situations where shared data is read often by multiple threads/ tasks and rarely written. A read-write lock is similar to a mutex lock except that it allows multiple threads to concurrently acquire the read lock whereas only one writer at a time may acquire a write lock. In the current scenario the *Insert* or *delete* operation on a list will require acquiring the read-write lock in the *write_lock* mode, while the seek (search) of a node will require acquiring the lock in the *read_lock* mode. By using the read-write locks we can have parallel search operations in the GED. The only drawback of using read-write locks is that locking operations take more time than the locking operations on mutexes. Hence locking strategy must be chosen carefully. Read-write locks are justified for the *event_paralist* and G_GED data structures in the GED where inserts to the data structures happen only once at the beginning when clients connect with the server; thereafter all other operations are search operations on the list to find a particular node. *Read_lock* mode can be used to allow threads to search the list in parallel.

Semaphore is a synchronization primitive that has a value associated with it, which is the number of shared resources regulated by the semaphore. Whenever a thread acquires a

semaphore, the semaphore count is decreased by 1. Whenever a thread releases a semaphore, its count is increased by 1. Any thread wanting to acquire the semaphore must wait till its count is greater than 0. Traditionally, semaphore operations have been known as P and V operations. P operation is equivalent to acquiring the semaphore (*sema_wait).* V operation (*sema_post)* is the same as releasing the semaphore. Semaphores are used primarily when there is more than one shared resource that needs to be regulated.

For synchronization of data structures in the GED, mutex locks or semaphores can be used when the operations involved are primarily inserts and deletes that require exclusive access. For data structures such as the *event_para_list*, where a majority of the operations are search operations on the list and updates on individual nodes, read-write locks can be used for locking the list and semaphore or mutex locks can be used for locking individual nodes. Details of the locking algorithm are explained in the implementation section. The table below shows the choice of locks made for locking the various data structures with reasons for the choice.

Table 1: Locks used for Lists and Log Files

DATA STRUCTURE and CHARACTERISTICS	LOCK USED with RATIONALE
Client_addr_list: list of client socket addresses. Nodes are inserted into the list when a client joins and are deleted when a client *unregisters* with the GED. Whenever the GED has to send a message over the socket to a client, this list is scanned	Mutex locks are used as operations are primarily inserts or deletes which happen when a client joins or leaves. These operations need an exclusive lock mode which is provided by the mutex lock. Since each node only has 2 fields, scan of list is a fast operation. Using mutex locks is preferred to read-write locks, also because operations on read-write locks have a high overhead.
Event_para_list: List of consumer event nodes. Nodes hold the events that are received by the GED and pulled by consumer clients. There is one node (*event_noti_node*) per consumer	Read Write locks are used for locking the list and semaphores are used for locking individual nodes as operations on the list are primarily search of the list to find an individual node, followed by updates on that node's contents. Shared mode (read lock) can be used while scanning the list to allow parallel scans and exclusive mode (write lock) is needed when nodes are inserted or deleted from the list.

	Semaphores are used for locking individual nodes as updates must be done exclusively.
Consum_list: list of consumers that connect with GED	Mutex locks are used as operations are primarily inserts or deletes when a consumer joins or leaves.
Event_ptr_l: list of event nodes.	Mutex locks are used as operations are primarily inserts or deletes. Since scans are very fast, mutex operations which are faster are preferred to operations on read-write locks.
G_GED: global event graph. This graph is traversed and parameter lists are propagated when event occurrences are sent to the GED.	Read-write lock for locking G_GED list. Write lock provides exclusive access to graph while inserting or deleting a node. When accessing list in shared (read) mode, lock hash table is used for managing access to individual nodes. Lock hash table minimizes number of semaphores needed to lock nodes of the G_GED. Thread suspend and continue calls are used to prevent more than one thread from accessing any node at a time. Lock table minimizes overhead of managing several locks.
Site_event_list: list of producers. There is one node per producer that holds the list of events that the producer must send to the GED	Mutex locks are used as scan operations are fast, and operations on mutex locks have a lower overhead than operations on read-write locks. The list is accessed only when a producer connects with the GED or recovers from crash, so operations on the list are also less.
Global log file (ged_spec.log). This file is used for construction of the G_GED graph when GED recovers from crash.	Mutex lock as file read or write must be mutually exclusive.
Event_counter: count of events received by GED.	Mutex locks as counter is constantly updated when new events arrive at GED

Consumer event log file	Each consumer event log file is locked separately by using Semaphore locks. Like mutexes, semaphores provide exclusive lock mode.

3.4 Improving I/O for Logging and Recovery

To provide recovery three log files are maintained by the GED, as explained in [11]. The consumer log file is given by *consumer_name.log* and is the file to which event occurrences sent to the GED are written before they are placed in the main memory buffer. In case of the crash of a consumer, events sent to the GED continue to be written into the consumer's log file. When the consumer client recovers after a crash, unconsumed events are read from the log file into the main memory buffers from where they are read by the consumer. Each log record has is identified by a unique LSN which is its log sequence number. At present the log file maintains two pieces of information in its header: *BLSN* is the log sequence number of the last event in the buffer. *DLSN* is the log sequence number of the last event that has been pulled by a consumer. The current algorithm for reading from the log file starts with reading the *BLSN* (during recovery it is DLSN). Thereafter, every record in the file is read until the record is reached whose LSN is greater than *BLSN*. After reading this record into the buffer, the *BLSN* value is updated. Thus, in order to read any record all the previous records have to be read first, which gets time consuming especially when the file is large. To speed up log file reads two new header fields are introduced. The file offset for the *BLSN* and *DLSN* can also be maintained as a part of the header. In this way read of a record will involve a read of its *BLSN* and *blsn_seek_offset*, followed by a *seek* into the file to the desired record. Records can also be read and written using the binary *fread* and *fwrite* which is faster than writing in ascii where *fprintf* and *fscanf* are used. Details of algorithm for speedup are explained in the implementation section.

CHAPTER 4
IMPLEMENTATION OF MULTITHREADED GED

As discussed in previous sections the RPC service request routines are threaded so that the GED can handle different procedure requests concurrently. In addition, some other procedures in the GED process are also threaded to give better performance, especially when the GED is running on a multiprocessor machine. This chapter discusses the details of the threading implementation. In the previous chapter, synchronization issues for the various GED data structures had been discussed. This chapter gives a detailed description of locking mechanisms used to synchronize access to the G_GED (global event graph) and the consumer event list data structures. In order to enhance performance of GED, changes were also made in mechanisms of buffer management and logging, which have been described in detail. Finally, the implementation of the *shut_server,* which was designed to gracefully shut down the GED is discussed.

4.1 Threading of RPC Procedures

The RPC library provides ***rpc_control*** function for multithreading purposes. It provides a framework for the server to create threads for the RPC function calls. When this *rpc_control* function is called with ''Automatic'' as the argument, a thread pool, whose default size is 16, is generated by the server dispatch routine *svc_run.*. When a request arrives, a thread from the pool will be activated to handle that request. Subsequent requests will be queued up if all the threads from the thread pool are busy. The following is a list of the RPC procedures:

global_reg : This thread is created for the RPC call where a client sends the name list of events (that it needs from other clients) to the GED, leading to the update of the Global event graph, G_GED.

namelist_update thread is created for the RPC call where the producer client updates its name list of events in order to know which events it should send to the GED.

global_notify thread is created for the RPC call where the client sends an event to the GED and the consumer event list (*event_para_lis)t* is updated.

receive_notify thread is created for the RPC call where the client pulls events from the GED by reading them from the *event_para_list*.

Unregister thread is created for the RPC call when a client unregisters with the GED.

4.2 Additional Threads in the GED

Additional threads were introduced to give a higher performance for the GED, especially when the GED is running on a multiprocessor machine. These threads are listed below:

1. *get_client_addr* is the handshake thread. The GED must continuously listen on a socket for incoming client connection requests. The handshake thread was introduced to listen for client requests on the socket. When the GED receives a message on the socket, it is parsed to determine the type of request. Requests are of the *init, resume, unregister* or *shut* types. In i*nit* mode client connects for the first time, if a log file already exists for the client, it is unlinked. In *resume* mode the client reconnects after a crash. Its socket ID in the client address list is updated. In *unregister* mode the client unregisters and will not reconnect in resume mode. The *shut* message is sent to gracefully shut down the GED.

2. *sendback_event_name_fun* thread updates the *namelist* for each producer site after the consumer notifies events of interest to GED. It updates the list of events for which the producer must notify to GED.

4.3 Locking of Global Event Graph (G_GED)

Global events are detected on the server using an event graph. An event tree is created for each composite event and the trees are merged to form an event graph for detecting a set of composite events. Leaf nodes represent global primitive events and the non-leaf nodes represent global composite events. For each node in the event graph there is an event subscriber linked list containing all the composite events that use this event as its constituent event. Each node, which is an operator, has a pointer to each of its child nodes for which it is an event subscriber. The parameter list *(L_OF_L_LIST)* is the list of event parameters. It contains the signature and values of arguments of the method that raised that event. When a global primitive event is detected at the leaf node, its parameter list is propagated to its subscribers, which are the parent nodes. By propagation we mean that the *L_OF_L_LIST* will be examined and copied or merged with another *L_OF_L_LIST* at the parent node, depending upon the type of composite event. Event occurrences flow upward to the root. If the composite event occurs for the current event notification, the composite event is detected. This detected composite event then further propagates upwards as a constituent of the composite event to be detected at its parent. Traversal

of the G_GED was discussed in chapter 2 in more detail. In a multithreaded server, several threads of execution share the G_GED graph, and access to the graph has to be synchronized. Using a mutex lock for the G_GED locking will give only two states (locked and unlocked) of access for the entire graph so that only one thread can be accessing it at any time. To give finer granularity, more than one thread should be able to access independent nodes of the graph concurrently, as long as they are not updating the same nodes.

One way to achieve a finer granularity would be to have a read-write lock on the global event graph and a semaphore lock on each node of the tree. However, when a large number of clients connect with the GED, the number of nodes in the tree will grow as each node represents a primitive or composite event or an intermediate node in the composite event tree. Allocating and maintaining locks for each and every node of the tree is cumbersome and will require too many locks. A better option is to maintain a hash table of the nodes of the tree that are currently being accessed. Each node of the tree will hash to a bucket of the hash table. The bucket will maintain a list whose elements represent the Ids of nodes of the G_GED currently being accessed. Thread IDs of threads waiting for a particular node will also be saved in a queue for each element in the list. In order to traverse the list of node IDs the bucket needs to be locked. This means that the maximum number on semaphore locks required for synchronizing access to the G_GED is equal to the number of buckets. In this way number of locks to be maintained is minimized and at the same time a fine granularity of locking is achieved for locking the G_GED.
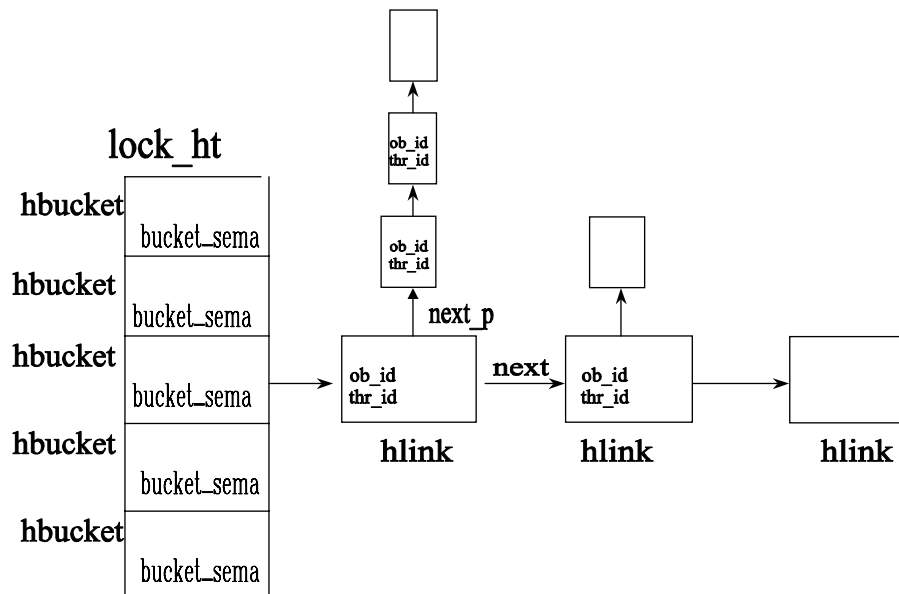

Figure 7: Lock Hash Table Data Structure

132

The classes defined for the hash table are *lock_ht , h_bucket* and *h_link. Lock_ht* is the lock hash table class. There is a single instance of this class for the GED. *h*_bucket is the class for each bucket of the hash table. Each bucket is guarded by a semaphore *bucket_sema*, and each bucket contains a chain *of h_link.* The *h_link* contains *obj_id, thr_id , next* and *nextp. Obj_id* is the unique ID (address of the node is used for hashing) of the G_GED node being accessed by a thread whose thread id is *thr_id. next* is a pointer to the next *h_link* in the bucket chain. *nextp* is a pointer to an *h_link* which contains the *thr_id* of a thread that is suspended and waiting to access the same G_GED node. Figure 7 gives the data structure of the lock hash table.

When an event occurrence is sent to the G_GED, the G_GED is traversed and parameter lists of one or more individual nodes will be updated. For traversal, read-write locks are used to give shared access to the G_GED. Lock hash table is used to access individual nodes when their parameter lists are being propagated. Each node of the G_GED must have a unique object ID for hashing purposes. Since the address of the node is unique, it is used as the object ID. The sequence of operations needed for locking are as follows: First the node object is hashed to find its bucket in the hash table. A semaphore lock (*bucket_sema)* is then acquired on the bucket so that no two threads may be accessing its chain of *h_link* at the same time. The bucket chain is then searched for the object ID of the G_GED node. If the object ID is not found, it means that no other thread is accessing this node. Then an *h_link* containing that node's object ID and thread ID are added to the bucket's *h_link* chain, the bucket semaphore is released, and the current thread is granted access to the G_GED node. The thread can now copy or modify the node's parameter list. On the other hand, if the object ID is found in the chain, it means that another thread is operating on the node at the same time. The current thread's thread ID is added to the list of waiting threads for that G_GED node. *bucket_sema* is released once the object ID is located, so that other threads can traverse the *h_link* chain for accessing nodes of the G_GED. The current thread is suspended and will be continued only when the desired G_GED node becomes available to it. After a thread finishes accessing the G_GED node, it removes its thread ID from the *h_link* chain. The next thread in the queue of suspended threads is released by a "*thr_continue*" and it can now access that G_GED node. Figure 8 gives the locking algorithm.
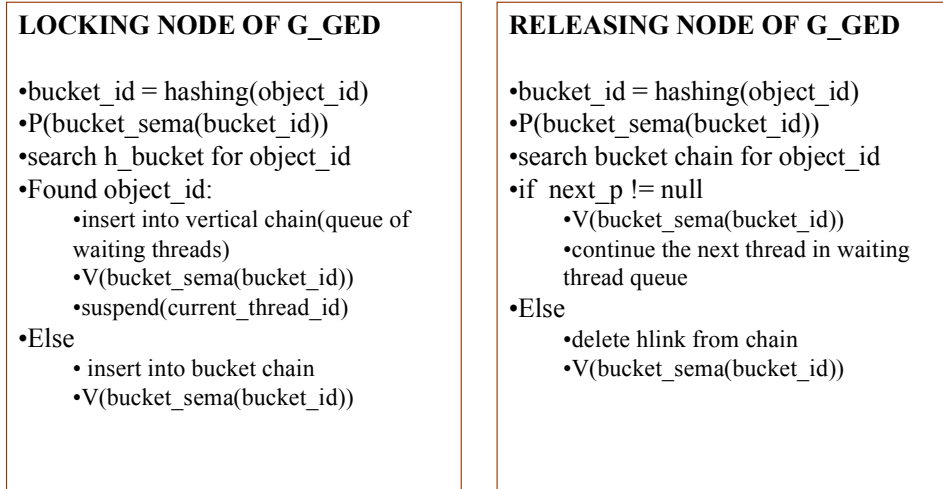
LOCKING NODE OF G_GED	RELEASING NODE OF G_GED

•bucket_id = hashing(object_id)
•P(bucket_sema(bucket_id))
•search h_bucket for object_id
•Found object_id:
    •insert into vertical chain(queue of waiting threads)
    •V(bucket_sema(bucket_id))
    •suspend(current_thread_id)
•Else
    • insert into bucket chain
    •V(bucket_sema(bucket_id))
	•bucket_id = hashing(object_id)
	•P(bucket_sema(bucket_id))
	•search bucket chain for object_id
	•if next_p != null
	    •V(bucket_sema(bucket_id))
	    •continue the next thread in waiting thread queue
	•Else
	    •delete hlink from chain
	    •V(bucket_sema(bucket_id))

Figure 8: Locking Algorithm for G_GED nodes

4.4 Locking of Consumer Event List

The *event_para_list* is the list of consumer nodes (*event_noti_node*) that carry event instances that are pulled by consumers. Figure 9 gives the data structure of this list. When producers send event instances to the GED, they are inserted into the list and they are pulled from the list by the event consumers. To have a fine granularity of locking and to provide maximum concurrency in access of the consumer event list (*event_para_list*), read write locks are used. As operations on the list are mostly read operations, having read-write locks for *event_para_list* will give maximum shared access to the list, and at the same time semaphore locks can be used for exclusive locking of the individual nodes when they are being updated. Sequence of operations for locking are as follows:

First time an event arrives for a consumer, create a new *event_noti_node* for that consumer. Then *write_lock* the *event_para_list* and insert the *event_noti_node* into the *event_para_list.*

Every time events are read from the *event_noti_node* (in *recieve_notify* call): Read lock *event_para_list*, seek to consumer's *event_noti_node* and release read lock. Then lock *event_noti_node* and copy its *para_list* for sending. Then initialize the *para_list* by *re_init().* Finally release semaphore lock on node.

When a client crashes: Read lock *event_para_list*, seek to the consumer's *event_noti_node* and release the read lock. Lock *event_noti_node*. Then initialize its *para_list* by *re_init().* Finally release semaphore lock on node.

Client unregistered: write lock *event_para_list*. Delete the *event_noti_node* for the consumer. Then write-unlock *event_para_list.*
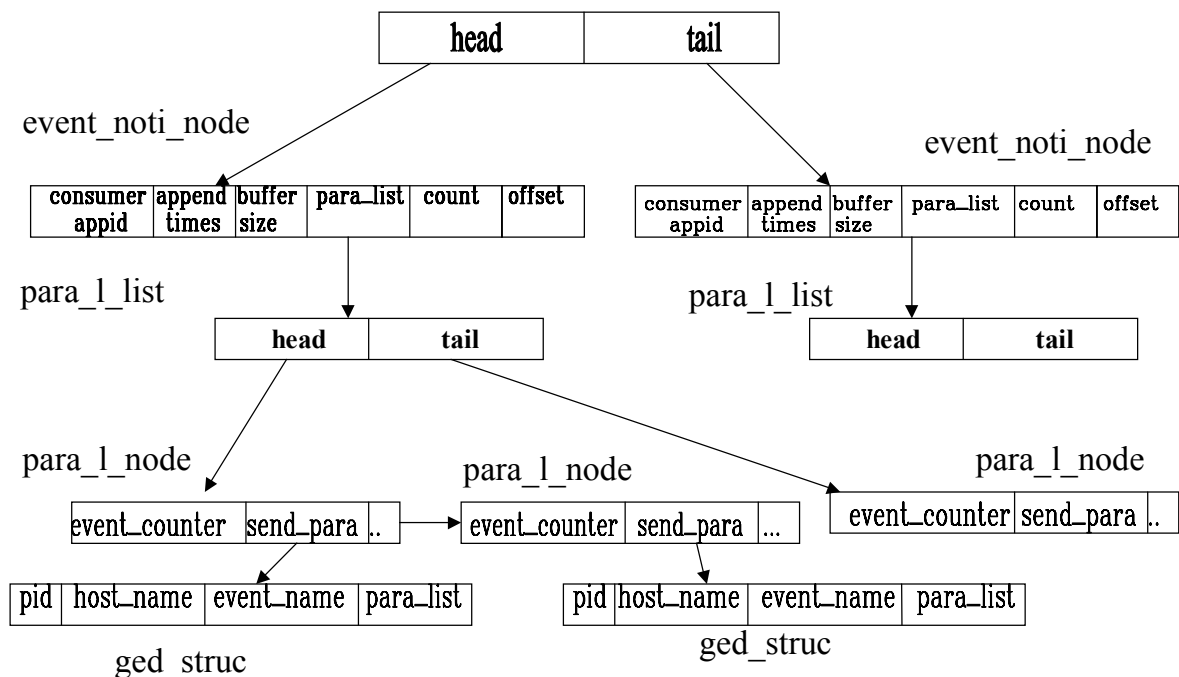
Figure 9: *event_para_list* Data Structure

Client receives event: The node is no longer deleted on a *receive_notify* call. Only its *para_list* is reinitialized. At this time a read lock is obtained to seek (search) the node in the *event_para_list* and then a semaphore lock is attained on the individual node while modifying its *para_list* via the member function *re_init()*. Finally semaphore lock on the node is released.

Client unregisters: write lock *event_para_list*. Delete the *event_noti_node* for the consumer. Then write-unlock *event_para_list.*

4.5 Performance Improvements in Buffer Management

Each consumer node (*event_noti_node*) of the *event_para_list* contains a *para_list* which is the list of event parameters. In the earlier implementation, every time the *para_list* was read from the consumer node through the *receive_notify* RPC call, the entire node was deleted from the list. Each time a new event was to be added, a new node had to be constructed and inserted into the list. This had an overhead of creating or destroying the entire consumer node each time it was accessed. This also added an overhead of locking the entire list for each insertion or deletion of the *event_noti_node*.

To improve concurrency, that is to reduce the number of times the list is write locked, the consumer node is inserted into the list *only the first time* a new event arrives for the consumer.

135

The node is no longer deleted when an event is read (*recieve_notify*) from the node. Only the node's *para_list* (which is the list of events within the node) is reinitialized by using a new function, *re_init()*. On a client site crash, the *para_list* is again reinitialized and the appropriate parameters (such as crash_flag) are set in the consumer node to signal a crash. Buffers held by this consumer are then released. However, the event_noti_node for this consumer is not deleted. This node is deleted only once when the consumer exits the scene with an *unregister()*.

4.6 Performance Improvement in Logging

For the purpose of persistence and recovery the GED uses a write ahead log.

Each consumer is assigned a unique log file given by *consumer_name.log*.

Event received by the GED for a consumer is first written into the consumer's log file and then inserted into the *event_para_list*. The log file serves two purposes. When the producer sends events to the GED at a rate much higher than the consumer's rate of event consumption, the extra events are read from the log file when the consumer becomes free. Each record in the log has a unique log sequence number (LSN) and the event parameters. Every event sent to the GED is first written into the log file and then inserted into the consumer's buffer, which is a node of the *event_para_list*. *BLSN* (buffer LSN) is the LSN of the last event in the buffer. If records need to be read from the file into the buffer, then the file needs to be searched for the record that has the lowest LSN greater than the *BLSN*. *DLSN* (delete LSN) is the LSN of the last record that was pulled by the consumer from the buffer. This number helps in recovering from GED crash. Every time the consumer pulls an event, the *DLSN* for that consumer is updated in its event log file. If the GED crashes, then all information of events in the buffer is lost. On recovery, it must read the *DLSN* in order to know which was the last event received by that consumer. BLSN is then made equal to the *DLSN* and events are read from the log file into the buffer as before. Similarly, when a consumer crashes, its buffers are freed after a certain fixed amount of events (*append_times*) are written to the log file and not pulled from the buffer. To recover from the crash, its *DLSN* is read and its *BLSN* is made equal to its *DLSN*, after which events with LSNs higher than *DLSN* are read into the buffers. Writing records into log files takes a considerable I/O time. In order to speedup reads and writes all data is written in the binary form using functions *fwrite* and *fread* instead of writing it in ASCII format using *fprintf* and *fscanf*. To speed up access of a record given its LSN, two new fields were introduced in the header of the event log file. Two new fields were introduced in the header: *BLSN seek offset* and *DLSN seek offset* with which it becomes possible to seek to the exact position of the record in the log file, given the *BLSN* and *DLSN* respectively. The header now has the following fields:

dlsn: log sequence number of the last event received by the consumer

blsn: log sequence number of the last event in the buffer.

blsn_seek_offset: offset position (from the start of file) where the BLSN record ends.

dlsn_seek_offset: offset position where the DLSN record ends.

The functions used for manipulating log files are *event_file_insert, event_file_to_list* and *event_file_delete.* These functions and their modifications are as follows:

event_file_insert: In this function an event is written into the log file. The open file descriptor and *L_OF_L_LIST* are passed to this function. Event information is appended to the end of the file. The entire *L_OF_L_LIST* is written into the file. Three structures: *copy1, copy2* and *copy3* were introduced for writing the list. Since the class *L_OF_L_LIST* contains several lists of unknown lengths, each time the length of a list is written first, followed by all the nodes of the list, which are of fixed length. While reading the list from the file, the length will be read first and then the nodes. Seek offset position from the top of the file where the inserted record ends is obtained using the function *ftell.* The function returns this offset position. This position information becomes a part of the *event_noti_node* if event is inserted into the *event_para_list*.

1. **event_file_to_list**: This function is used for reading events from the log file into the buffer. When a producer sends events at a speed faster than the speed at which a consumer receives them, the consumer buffer gets full. Write ahead log makes sure that the events are persisted, whether the buffer is full or not. At a later point, when the consumer removes events from the buffer, events that could not be stored in the buffer are read by the GED from the log file into the buffer and the *blsn* and *blsn_offset* value in the file is updated to reflect the LSN of the last event read into the buffer. When a consumer crashes, it no longer receives its events. After detecting the crash the GED frees its buffers and thereafter, events for this consumer are only written to the log file. They are read from the log file once the consumer recovers. When the GED fails, it may have some events in the consumer buffers, which were not yet consumed by the consumers. On GED recovery, the *DLSN* is used to read unconsummed events from each consumer's log file. For each consumer, the *Event_file_to_list* function is passed the open file descriptor and the offset position in the consumer's log file from where the event is to be read. Data is read into structures. After reading an event into the buffer the *BLSN* and *blsn_seek_offset* are updated to reflect the last event read.

2. **Event_file_delete**: When the consumer removes an event from the buffer, the log sequence number and seek offset are read from the *event_noti_node*. These are used to update the

DSLN and *dlsn_seek_offset*. On the crash of a client or GED failure, *BLSN* and *blsn_seek_offset* are made equal to *DLSN* and *dlsn_seek_offset* respectively.

4.7 Design of Shut Server

A *shut_server* has been implemented to shut down the GED. Host and port on which the GED is running are passed as arguments to the *shut_server*. The *shut_server* establishes a connection with the GED process on a specified port and sends a "*shut*" message over the socket to the GED that is listening for client requests in the *get_client_addr* thread. Command used to shut the server is:

shutserver –m[ged machine name] –p[port name]

On receiving the message, the GED parses the request to determine the request type. The GED then goes through a clean up procedure before exiting. In the clean up: 1) GED_spec.log, client_addr.log and each of the consumer event log files are deleted. 2) *.recover* file (used to detect GED recovery from crash) is deleted. 3) memory is freed for

the buffer manager and data structures. In this way the GED gracefully exits by releasing all its resources.

CHAPTER 5
PERFORMANCE EVALUATION OF THE ENHANCED GED

This Section outlines the tests done to measure performance enhancements in the multithreaded GED server. Earlier the GED could handle only one RPC request at a time. When multiple clients connected to the GED and sent concurrent requests, the requests were effectively serialized. With the multithreaded implementation the GED can handle several requests concurrently in separate threads. This is expected to effectively reduce the response time, which is the difference between the request made by a client to the reply received by it from the server.

Profiling a program is a meaningful approach to identifying its performance bottlenecks. To track the time that the server spends using the CPU or waiting for locks and I/O completion, the profiling tool *prof* available on UNIX was used. Throughout the testing of the GED implementation, the *prof* tool was used to look at the times spent by the various operations in GED and to make adjustments in the program. For example, it was seen using the *prof* tool that in the server process a significant time was being spent in waiting for the lock for the event log files. To reduce this wait the lock allocation was changed from one mutex lock for all the event log files to one lock per event log file for each consumer. This reduced the response time at the consumer as each consumer log file could be locked separately and locking was independent of other event log files so waits were reduced.

To set up test programs for experiments we create a specialized producer client program that can send the server a stream of events and measure its response. A consumer client program is also created that subscribes for the same set of events with the GED and has rules defined on them. The test client measures the total time it takes to complete a large number of event notifications. For the producers these operations are the send of event occurrence to the server; for the consumer the operations are primarily the receive (pull) of events from the server. For the experiment, multiple client processes issue requests to the server across multiple connections.

To evaluate multithreading, we run the GED server in two different modes – a serial server (one that does not use threads at all – runs in single threaded mode) and a multithreaded server. The

GED server threading mode can be specified in the configuration file before starting the server. When running in the multithreaded mode, the number of threads in the thread pool to handle the requests is also varied. The server is first run in a single threaded mode and number of threads are then increased to 16, 32, 64 and 100.

The *persist* or *nopersist* mode is another factor that affects the performance on the GED server. In the persist mode the GED server uses a write ahead log to provide client as well as GED recovery. It writes every event received from the producer into a log file using the function *event_file_insert*. Records may be read from the file using the function *event_file_to_list*. The main difference in the two modes is that in the persist mode the request service procedures are I/O intensive. Another major difference is that in the *nopersist* mode there is no buffer manager to hold the excess events. Every event received from a producer has to be sent to the consumer of the event before the next event for that consumer is accepted. In other words in the *nopersist* mode the producer cannot send at a rate higher than the receiving rate of the consumer, and procedure calls made by producers or consumers are effectively serialized. In the persist mode, the rate of send by the producer is independent of the rate of receive.

Performance of the GED depends on the number of clients and contention. The type of events that the consumers subscribe to affects contention. For example, if we have a set of consumers C1, C2, C3, C4 that subscribe to a set producers P1, P2, P3, P4 respectively, where each producer generates a different event, then all the 4 consumers can be consuming events in parallel with no sharing as each will be accessing a different event buffer of the consumer event list. Contention will be less in this scenario. On the other hand if all the four consumers C1, C2, C3, C4 subscribe to the same event from a single producer P1, contention is high. The number of clients being serviced by the GED also affects contention. As the number of clients connecting with the GED increases, the contention increases.

The speed of event generation by the producer clients also affects the time difference observed between single and multi threading modes. If the producer generates events with a significant delay between each event (1-2 seconds) then this interval is much more than the time required for processing the event. Therefore the response time will almost be the same, whether the GED is run in the single or multithreaded mode. On the other hand when events are generated with no delay between them response times will vary for the single and multithreaded modes.

5.1 Experimental Setup

In order to have multiple clients, tests were run with 4 consumer and 4 producer clients that connected with the GED. Tests were performed on an 8-CPU multiprocessor machine. All clients were started together and connected with the GED for each set of readings. To get an accurate measure of readings, 10 readings were taken for each test case scenario by running it 10 times and the average was calculated. In order to have a large number of events to the GED, each of the 4 producer clients was made to generate a 100 events. Since the buffer size of the event buffer was kept low to 5 buffers, a 100 events by each client was a sufficient value to test performance of the buffer manager under heavy load. Different sets of readings were obtained when running in the persist mode as the number of threads was increased from 1 to a 100. A timer was started at each client when it received the first event and stopped when it finished receiving the last event. The time value obtained was divided by number of events to find the response time of a single RPC call.

The server's response to a client's request involved different amounts of I/O and more or less CPU intensive tasks. The different times measured at the client were:

Response time: Time between the request and reply at the client. This was used to measure the performance gain. *System time* is time spent in CPU intensive system tasks. *User time* is time spent in I/O. *CPU time* was the total of *system* and *user* time. The timing functions used were *gettimeofday* to measure the response time, and *getrusage* to measure the CPU times. The readings obtained for the different scenarios are shown below.

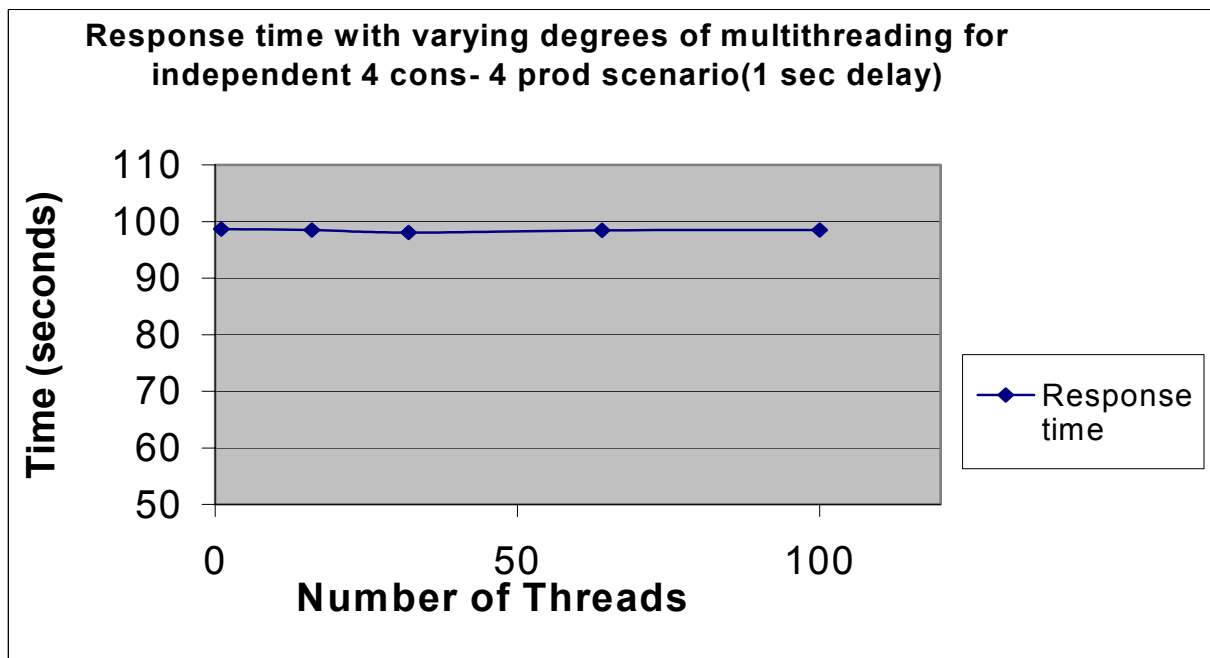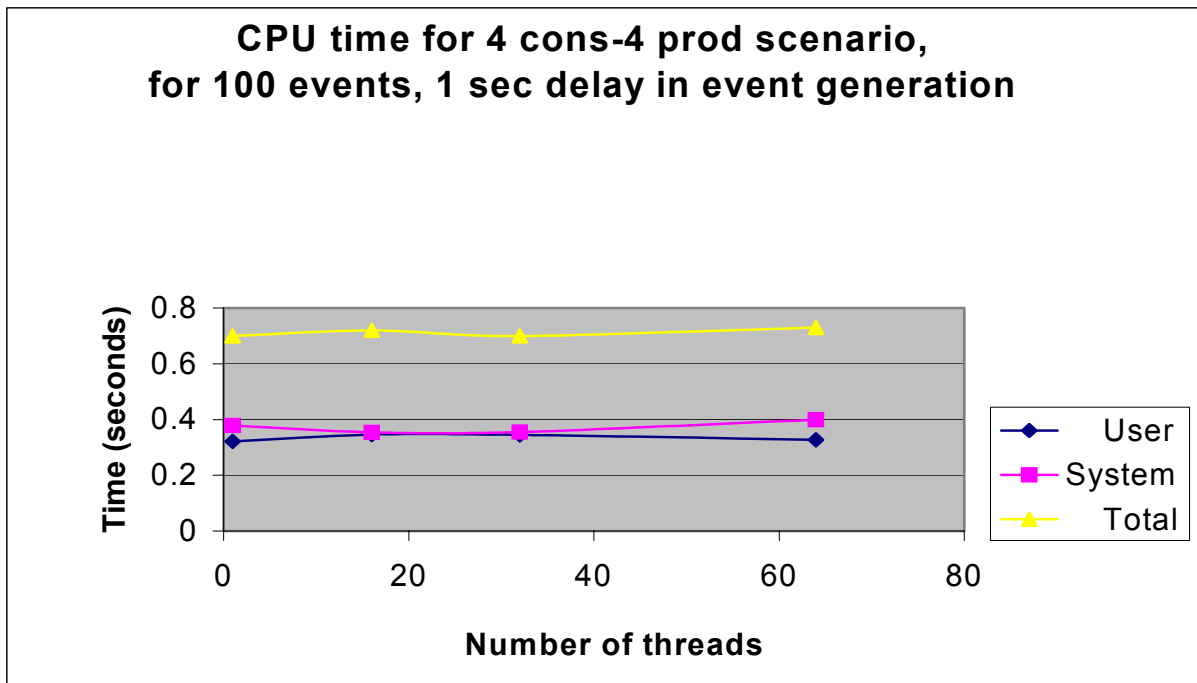| Threads	User	System	Total	Response time
1	0.3214	0.3786	0.7	98.6587
16	0.346	0.354	0.72	98.52
32	0.345	0.355	0.7	98.081
64	0.327	0.4	0.73	98.45
100	0.338	0.362	0.72	98.52

Figure 10: Time Measurements for 4 prod- 4-cons (1 sec delay in event generation)

For 4 producers and 4 consumers subscribing to independent events, the results are shown for 1 second delay between event generation in figure 10. These readings are taken in persist mode. The total CPU time remains almost the same even if the number of threads is changed because the total amount of CPU power needed for processing is the same. The GED has a certain capacity for processing events. It takes a certain maximum amount of time for the processing. If the delay between the events sent to the GED is more than this maximum then even if the GED is run in multithreaded or single threaded mode the processing will always be done within this time. Since the delay of 1 seconds is more than the time required for processing an event, the response time doesn't change much by changing the number of threads for this test case.

In the 4 consumer- 4 producer case (Figure 11), when there is no delay in event generation with consumers C1, C2, C3, C4 subscribing to events P1, P2, P3, P4 the response time shows a drop at the first reading for the multithreaded case (16 threads). This is because when the 16 threads are spawned to handle the request, processing can be done in parallel. As the number of threads is increased, contention and wait time for locks increases and is an overhead, thus response time again increases for 32 threads. Increasing number of threads beyond a certain point does not improve performance further as the CPU power available for the process cannot increase.

For the 4 consumer-1 producer case, the response time drops initially upto 32 threads and then becomes steady as increasing number of threads will not improve performance further; As opposed to the 4consumer – 4producer case, where there can be 8 or more threads trying to update the buffers at anytime, in the 1 producer- 4 consumer case there is only 1 producer that will be adding events to the buffer. The 4 consumers can be consuming events in parallel. As contention is less in this case, response time becomes steady even as the number of threads is increased. Figure 12 shows the time measurements for this scenario.

Figure 11: Time Measurement for 4prod-4cons (0 sec delay in event generation)

Threads	Response time
1	17.9912
16	13.2475
32	16.552
64	15.4125
100	16.09

Response time for varying degrees of multithreading for independent 4 cons-4prod scenario (0 sec delay)
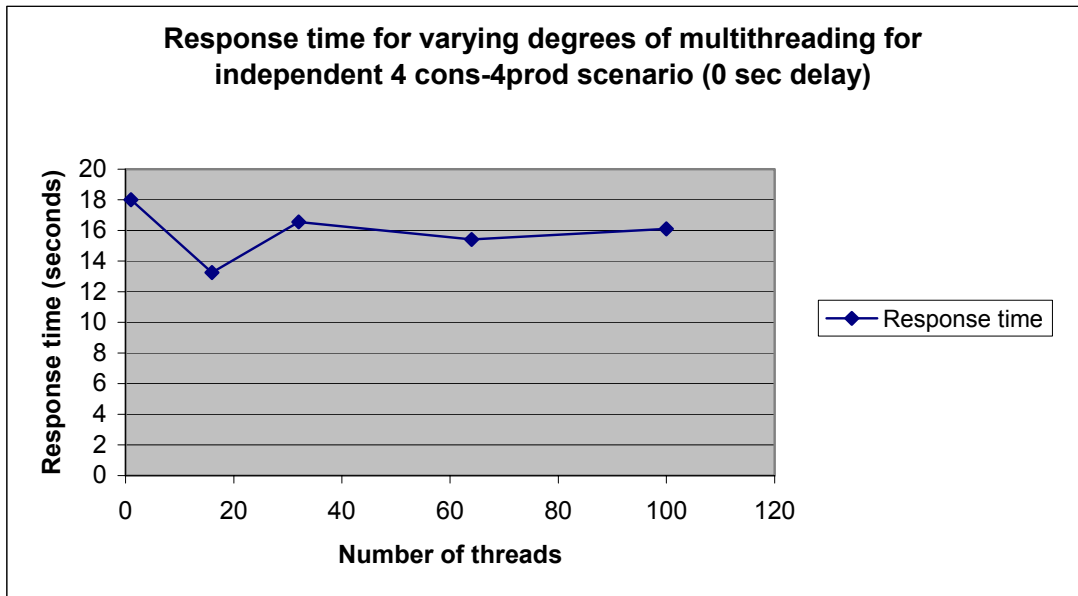
Figure 12: Time Measurement for 1prod-4cons Scenario

Threads	Response time
1	20.1
16	18.94
32	14.93
64	13.45
100	13.56

**Consumer Response time for
1prod-4cons case (0 sec delay)**
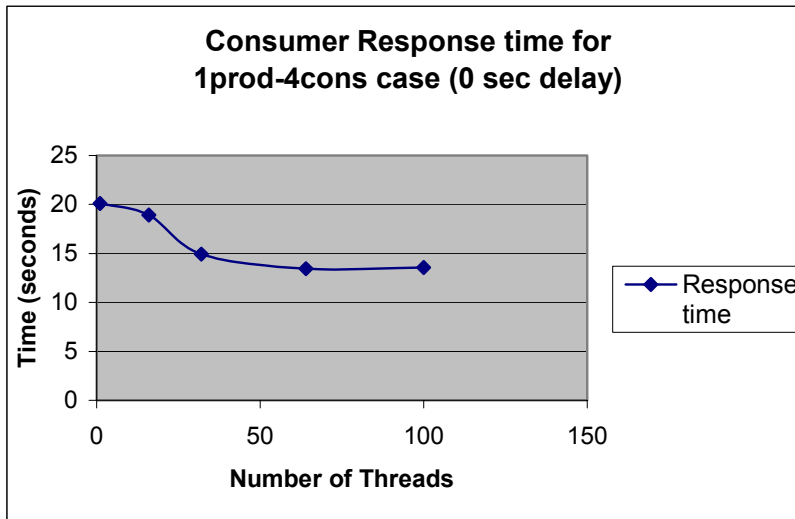
As shown in figure 13, producer clients benefit the most with the threaded GED. This is seen by measuring the producer's response time (time between the point when the producer starts producing the first event to the time at which it returns after finishing sending the last event to the GED). GED is running in persist mode on eclipse which has 6 processors and the producer is generating events for 4 consumers. The producer generated 100 events in a second The response time drops heavily when there is a switch from the single to multithreaded mode (16 threads). This is because the producer generates all the events at once; of the 16 threads created, up to 3 may be used by the consumer in making an RPC call to pull the event. The rest of the 13 threads are all available to the producer for doing the event notification in parallel. Time measured for single threaded case was 21.5 seconds. If this time is divided into 13 threads, then a fall to 1.9.seconds is expected. The response time for multithreaded case remains around 2 seconds as expected and gets steady as the number of threads are increased because the total processing power allocated to the server process is constant.

Figure 13: Time Measurement for prod Application

Threads	Response time
1	21.5
16	2.705
32	2.44
64	2.46
100	2.35

Producer response time for 100 events (0 sec delay in event generation)
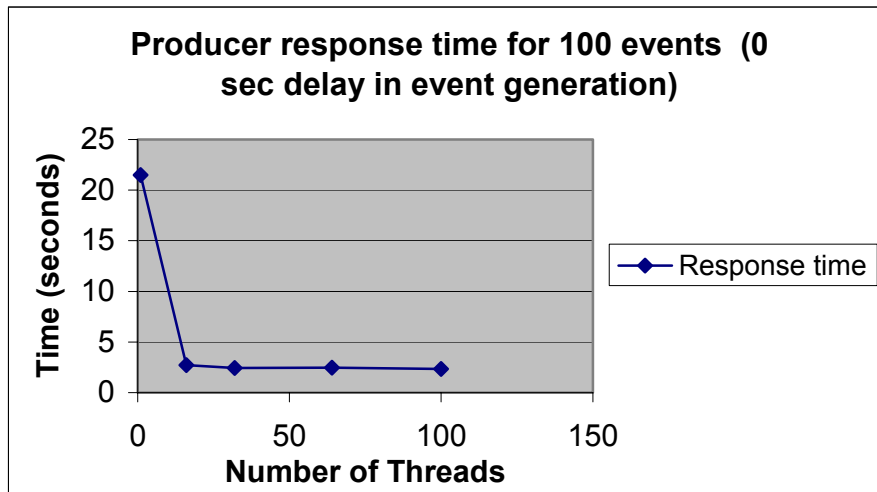
5.2 Summary

The results obtained in the above test scenarios (Figure 10-13) show that improvements measured in response time on switching from single to multithreaded case, are most significant in the cases where consumers subscribe to independent events. The benefits measured in terms of response time are best seen by a producer of events who can generate all its events and continue to do other useful work without being affected by the speed at which events are received by the consumer. For consumers a drop in response time is observed in moving from single to multithreading modes. When events are generated with 0 delay by producers, rate of generation is higher than speed of consumption and events may have to be read from log files. In this case the number of also I/Os also increases the response time measured at the consumer.

146

CHAPTER 6
DESIGN ISSUES FOR RULE SUPPORT

In the current implementation, the GED is merely an event mediator, passing global events across applications. This means it can be used readily for purposes such as stock applications where the price decrease in IBM stock causes a rule to be triggered in the stock application, and this event is propagated without any delay by GED to a "stockbroker" application. "Stockbroker" application fires a rule to purchase the stock immediately at that price.

If the GED was to be used in situations such as data warehousing, where data from various heterogeneous sources is collected, filtered, and sent to a warehouse for integration, capabilities of the GED have to be enhanced. GED has to be augmented to do certain useful computation on the received global events or delay the propagation of events to the consumer applications. Thus a design goal is to add the capability of rule support on global events at the GED. Supporting rules will involve 1) design of a rule editor for rule specification, 2) back end rule server for rule persistence and management, and 3) a dynamic loader for loading rules into the GED address space at runtime.

6.1 Extensions to the Graphical User Interface

To meet the requirements of multi-platform usage and portability, Java has been chosen as the implementation language for the interface module of the editor. At present a rule editor interface exists for specifying rules on events local to applications. This interface is extended for specification of rules on global events. The opening window will now allow the user to select between a choice of global and local rules. Based on whether Global or local rules were selected, appropriate chain of windows will follow to give the user convenience in editing rules.

6.2 Architecture

Server architectures can be categorized as 2-tier or 3-tier based on how the client/server application can be split into functional units. The typical functional units are the user interface, business logic and the shared data. In two-tier client/server systems, the application logic is either buried inside the user interface on the client or within the database on the server. A three-

tier architecture augments traditional client/server computing by introducing a middle tier component [12]. There are three component layers: The front-end component which is responsible for providing portable presentation logic; the middle-tier component, which has the application logic and allows users to share and control business logic by isolating it from the actual application; and the back-end component, which provides access to dedicated services. Three-tiered client server architecture is used for the rule editor. The main reason why a three tiered architecture is chosen is that the processing of the requests coming from the interface often gets complex and is best performed on a dedicated application server rather than by the database server. Three tier applications are more scalable because they minimize the load on the server. Extending a three-tier server is also less complicated than extending a two-tier server.   The user communicates with rule editor at the topmost layer. At the middle layer there is a rule server, similar to an application server which will do the processing or the rules, persist them into files and service requests from the editor (client to rule server) for retrieval, modification or deletion of rules. The rule server will also compile the condition and action functions associated with a rule and archive them into libraries. At the lowermost layer is the file system where rules are persisted. A dynamic loader is an additional unit needed for finally bringing persisted rules into memory in order to load these rules on the GED by calling EVENT and RULE constructors at GED runtime.
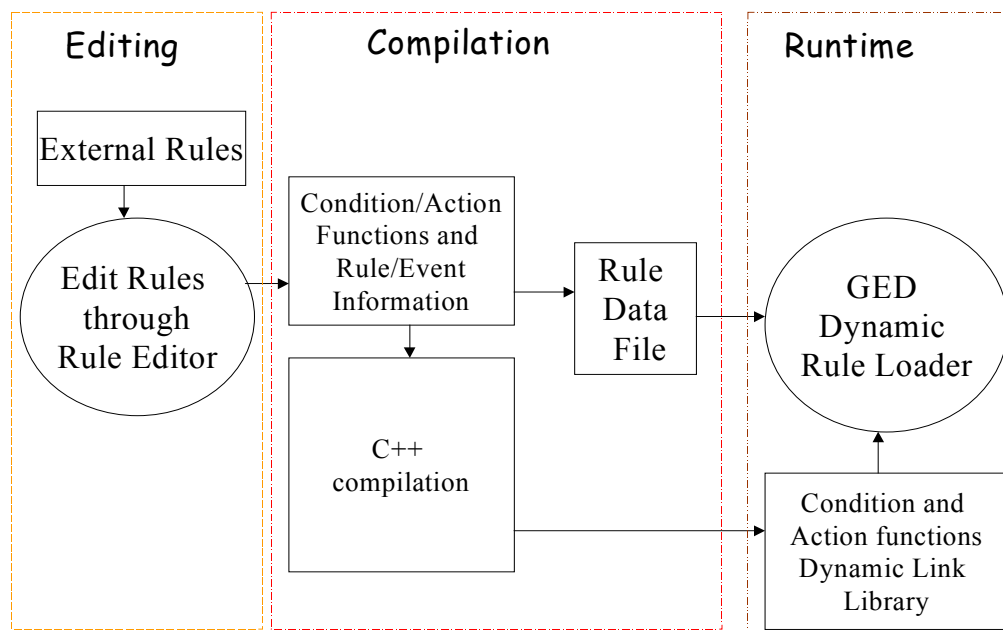
Figure 14 shows the different phases of rule creation.

Figure 14: Phases of Rule Creation

6.3 Rule Persistence

Sentinel applications use Exodus Storage Manager to store and manage objects. Exodus is also currently used as the storage manager of the rule database, for persisting rules defined through the rule editor interface. Exodus storage manager stores all data on volumes which are either UNIX files or raw disk partitions. The Object Query Language (OQL) is used for accessing data stored in the database. Using Exodus makes the GED dependent on OODB and Exodus. To make the GED independent of Exodus, rule information needs to be persisted separately in files by the rule server. Strings of information received by the rule server from the rule editor need to be written to files in a format that is efficient and convenient to read.

6.4 Dynamic Loading of Rules

In order to dynamically load pre-composed rules onto the GED, a function named *load_ged_rules* will be called at GED start up time before it does handshakes with new clients. This function retrieves the persisted rules, events, condition and action names. It then associates rules with their respective events and creates lists of event and rule objects in memory. It then traverses these lists and calls appropriate RULE and EVENT constructors for creating rule and event objects. On traversing the lists of rule objects it will read the condition and action names and invoke the dynamic loader to dynamically load the condition and action functions for the rules. The events and rules will finally be inserted into the G_GED global event graph. Details are explained in the next section.

6.5 Portability

The rule server needs to be portable and will thus be implemented in Java.

Java has security, network, and machine/ platform independent features.

Java is an object-oriented programming language with syntax is similar to C and C++ that are "main-line", industry-proven languages.

Java has been integrated with the Word Wide Web (WWW). By using Java as the language and by embedding Java programs in the form of applets into HTML pages, the program can also be accessed from the web.

The following chapter discusses implementation details of the modules for rule support.

IMPLEMENTATION OF DYNAMIC RULE EDITOR SERVER

The previous chapter discussed design issues for the three-tiered rule editor server architecture. This chapter describes the implementation details of its modules. Extensions that were made to the existing graphic interface to support global rules are discussed first. The interface communicates with the back end rule server by sending messages in ASCII in a certain format. Summary of the messages and their meanings are given next. The rule server was written in Java and is made up of *RuleEditor* class and *HandleConnection* classes. The purpose served by each class has been described. When new rules are defined through the interface, the server archives compiled condition and action functions into a library. When a rule is committed, rule information is also persisted into a rules data file. Format of the data file, followed by the details involved in reading the data file and construction of an event graph in memory to load rules on the GED at its runtime are the final sections of this chapter.

7.1 Extensions to the Rule Editor Graphic Interface

Abstract Window Toolkit, commonly referred to as the AWT is used for the graphical interface. The following 4 classes make up the functionality of the toolkit: *Component* class, *Container* class, *Graphics* class, and *LayoutManager* interface. The AWT is a platform-independent windowing toolkit. The AWT delegates the actual rendering and behavior of frames to the native platform-dependent windowing system.

In the extended interface the user is given a choice of defining rules on (1) Global events or (2) local events. If the user chooses global he can either define his own group or choose from a set of groups already formed. Each group is made up of a set of applications. Files belonging to each group are in a separate directory in the file system. Once the group is chosen, the editor displays the global events belonging to the applications within the group. User can define rules on these global events. Rules can be defined on global primitive events or on composite events which the user will compose using the interface. The user, through the interface, also defines the condition and action functions that are sent to the back end rule server where they are compiled and

archived into a library. Results of compilation of condition and action functions will be sent back to the interface where messages are displayed. The user can then reedit the functions or follow a sequence of windows to define rule name and rule specific information such as context, priority, coupling etc and finally commit the rule. When rule information is committed, it will be persisted by the back end server and will be reloaded at GED runtime.

7.2 Message Driven Services

In order to communicate with the back end rule server, the interface connects with the rule server on a specified host and port using a socket. Information from the editor interface to the rule server flows in the form of string messages over the socket.

Following is the summary of messages that can be sent by the interface to the rule server to request a specific service:

"Ggroup\n" : This message asks the interface to send names of global events of applications belonging to the specified group. All files belonging to each group are placed in a separate directory given by the group name. The Rule editor server reads the *Global_signatures* file belonging to that group to extract the appropriate global event information from it. It then sends this information in the form of strings to the editor, where it will be displayed in a menu window.
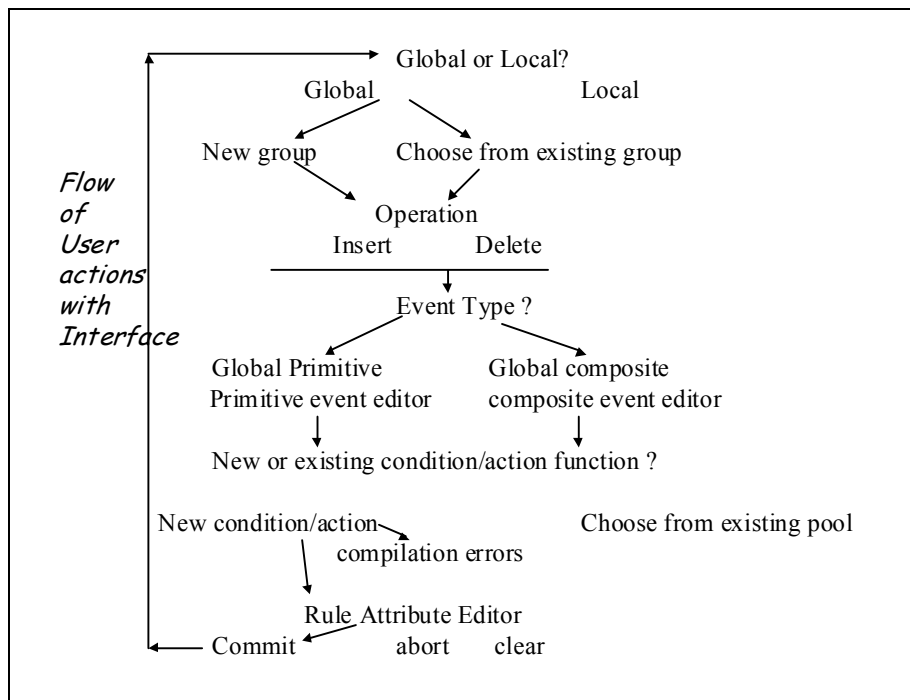

Figure 15: Flow of User Interaction with Interface

151

"$group\n" : Send the name list of the given group's existing condition functions to the rule editor interface. The information is read from the rules data file.

"~group\n" : Send the name list of the given group's existing action functions to the requesting client. The information is read from the rules data file.

"*compile_functions\n*" : Compile the given functions in the temporary directory of the given group and send the results of the compilations to the requesting client.

"commit\n" : Add the given condition/action functions to dynamic linking library (if they are new), move the source files of the functions to function pools belonging to the group (if they are new), and persist the given rule information in the rule log file.

"commit_delete\n" : Remove the given condition/action functions from the dynamic linking library space (if they are not shared), remove the source files of the functions to function pools belonging to the group, and remove the rule information from the rule log file based on the given type identifier.

"abort\n" : Clean up the given functions (both source files and object files) in the temporary directory.

7.3 Server Classes

The rule server is written in Java and has two main classes: RuleEditorServer class and HandleConnections class. **RuleEditorServer class** takes a connection request from the editor interface and then spawns a new thread to handle the connection. Since Java is a system independent language, the program cannot directly access environment variables. While starting the server, environment variables such as path of the sentinel system directory, which are used in accessing the various data files, must to be made available to the program. This is done by running a shell script, which loads the environment variables into the *System.properties* structure that can be accessed by the Java program. TCP/IP sockets are used for communication with interface. Socket object is an instance of *ServerSocket* class provided in the Java API. Once the interface establishes a connection with the rule server, string messages are exchanged between the two processes and processing is handled in the *HandleConnection* class.

HandleConnections class invokes the method *handle()* to service the requests such as persist, retrieve, or modify rules which are sent by the interface. Based on the type of message the request is processed and results are sent back to the interface. To do system dependent tasks, such as use *make* on the Unix system, the server uses *exec* command to fork a new process that runs a Perl script to do the operations.

7.4 Rule Persistence

Since the definition of rules through the interface and the loading of rules on the GED server does not happen at the same time, rule data needs to be persisted by the server so that rules can be reconstructed at GED runtime. Whenever a rule is committed through the interface, information needed to construct the rule is written into a rules data file. Each record contains information of a single rule. Each record ends with a field called *Valid.* When a rule is deleted, the *Valid* field is set to 0. When rules are read from the file, records whose *Valid* field equals 0 are ignored. Following is the format of a record in the Rules Data file:

Opcode   evnt_name   send_back   site   event(type) con_evnt1con_evnt2 con_evnt3   rule_name   priority   coupling   context   trigger   Valid

Opcode is a numeric value that gives the type of the global event. For example, a global primitive event has an *opcode* of 0, composite event "*and"* has an *opcode* of 1 and the composite event *"or"* has an *opcode* of 2. *event_name* is the name of event on which the rule is defined. *send_back* is a numeric value that determines whether the event given by *event_name* will be propagated back to the consumer. This field is always set to 0 in the rules data file as the EVENT node will only be constructed to fire a rule at the GED site and the event occurrence does not have to be sent to any consumer client. At a later stage when the GED is running, if a new client connects with the GED and registers for the same event, then the value of *send_back* will be made to 1 in the EVENT node in memory. *Site* is the name of the *send_back* site. Its is given a dummy value as there is no consumer that has subscribed for this event at the time of Rule creation. If the event given by *event_name* is composite, then *con_event1* is the name of its first constituent event. *con_event2* is the name of second constituent event. *con_event3* is the name of the third constituent event in cases of events such as *A* or *A** or *NOT*. *rule_name* is the name of rule. Context, priority, trigger and coupling give the values of the respective parameters associated with the rule. *Valid* is a numeric field that determines if rule has been deleted.

7.5 Dynamic Loading of Rules on the GED

In order to make the GED capable of supporting rules, a RULE class was incorporated into the GED code. An object belonging to the RULE class inherits from NOTIFIABLE and REACTIVE classes in the GED class hierarchy. A RULE object is *notifiable* because it must be notified when the corresponding event occurs. At the same time, it is reactive because the action of the rule object may raise an event that triggers another rule, leading to nested execution of rules. When the GED is started, rules data that was persisted into the log files must be loaded into memory to create RULE and EVENT nodes that form the Global event graph. This is done by the *load_ged_rules()* function. To prevent the rule and event nodes from being duplicated, a main memory data structure (*EventList*) is created first from the data read from the file. Construction of the *EventList* structure in memory also makes the construction of composite EVENTs more efficient. It prevents the need to again search the rule data file for complete information about a component event that makes up a composite event. The rules data file has to be read just once and all the information about the component events is made readily available in memory for the construction of EVENT nodes. Since every rule is related to a specific event, a *RuleNode* is inserted into the rule list of its respective *EventNode.*

The Following Classes were defined for the creation of graph in memory:

RuleNode contains rule information such as rule name, context, priority, trigger mode, condition and function names.

RuleList is the list of rule nodes. Each event node contains a list of rules to be fired on occurrence of that event.

EventNode contains event information such as event name, site name, component events (in case of a composite event) and a rule list, which is the list of rules defined on this event. Whenever a new rule is defined on this event, it is inserted into its rule list.

EventList is the list of Event nodes. Figure 16 gives the *EventList* data structure.

After reading each record from the file, a *RuleNode* is constructed. If the Invalid field of a record is 1 then no *RuleNode* is constructed for that record. If an *EventNode* does not exist for the corresponding event, then it is also constructed.

154

Figure 16: EventList Data Structure
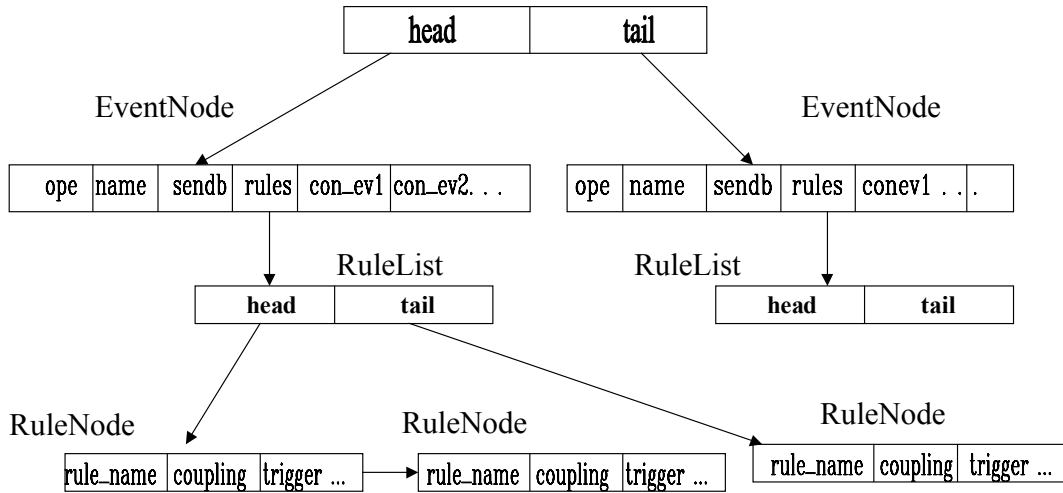
To construct a composite *EventNode*, the *find()* operation is used which searches the *EventList* based on the constituent event name and returns a pointer to the constituent *EventNode*. Therefore each composite *EventNode* has pointers to its component *EventNodes*. After checking for duplicates, a *RuleNode* is inserted into the *RuleList* of the *EventNode*. New *EventNodes* are inserted into the *EventList*. After all records from the rules data file are read, construction of the *EventList* is complete. The *EventList* is then traversed for construction of the EVENT and RULE nodes, which make up the G_GED global event graph.

The library *GlobaldynCondActLib* contains the object code of the condition and action functions that had been compiled in the preprocessing and compilation phase. While loading rules on the GED, this library is first opened (using *dlopen*) and the library handle is obtained. After the *EventList* is built in memory the function *create_events_rules*() is called on the *EventList* to construct EVENT and RULE nodes. The *create_events_rules* function takes the library handle as its argument. EVENT constructors are called for those nodes in the *EventList* that have a nonempty *RuleList*. While creating the EVENT nodes, the *config_name_list* is also checked. This list contains the mappings from the machine names used at compile time to the machine names used at actual GED runtime for the site which produces the event. Before creating any EVENT node, the G_GED graph is searched for a node with the same name by using the *get_prt_comp()* function. If the EVENT node is already present in the G_GED, it need not be

constructed. This prevents creation of the same node multiple times and saves memory. For each *EventNode*, the *RuleList* is then traversed to create RULE objects which *Subscribe()* to the EVENT. While constructing the RULE object, *dlsym* call is made to the dynamic linking library. The *dlsym* call takes the condition or action function name to dynamically load the object code for the condition or action function into the memory. The following is the code from the body of "*create_rule"* method present in the *RuleNode* class. A RULE constructor is being called below:

int (*fptr_cond)(L_OF_L_LIST*)=NULL;

void (*fptr_act)(L_OF_L_LIST*)=NULL;

fptr_cond=(int(*)(L_OF_L_LIST*))dlsym(dyn_lib_handle,condition_name);

fptr_act=(void(*)(L_OF_L_LIST*))dlsym(dyn_lib_handle,action_name);

…..

RULE *rule_ptr = new RULE(rule_name, event_ptr, fptr_cond, fptr_act, coupling, context);

………

………

In this way RULE objects are constructed and loaded on the G_GED. The *site_event_list* is a list maintained at the GED that has one node per producer. Each node of the list contains the potential events that the producer must send to the GED. Every time an EVENT node is constructed for defining a RULE, the producer's list (*site_event_list)* has to be updated so that a producer connecting with the GED will know that it must send occurrences of this event to the GED. This is done by calling *update_site_evnt_list()* function. When GED handshakes with clients that are producers, they will read the *site_event_list* to know which events to send to the GED. For each event occurrence that is sent to the GED, the rule condition will be executed and if it evaluates to true, the rule will be fired.

CHAPTER 8
CONCLUSION AND FUTURE WORK

8.1 Conclusion

This thesis extends earlier work on the Global Event Detector in Sentinel to make the GED scalable and to improve its performance. A multithreaded architecture for the GED was proposed for increasing its performance and scalability. Synchronization issues related to multithreading were addressed. A lock hash table was implemented for handling locking of the Global event graph and read-write locks were used appropriately for locking the consumer event list that is also a heavily accessed data structure in the GED. Performance of the recoverable GED was also improved in logging and buffer management by implementing a better format for the log files and by changing the algorithm to deal with handling of events in the buffer. Performance measurements show that multithreading resulted in a reduction in the response time for the producer as well as consumers of events. This thesis also expressed the need for rule support and implemented a rule server that allowed the GED to fire rules on global events at its site.

8.2 Future Work

GED can be made hierarchical so that the GED can serve both as a server to monitor events for clients, and as a client that will register for events from other servers.

We can have a hierarchy of GEDs to support real life applications where filtering of data is taking place. A localized group of applications can be at one hierarchy. The hierarchy may reflect a common chain of organizations.

Replicated GED: Clients connect to more than one GED but send events to either one or both (depending upon whether the global event space is partitioned or replicated).

REFERENCES

[1]   L. Hyesun, Support for Temporal Events in Sentinel: Design, Implementation, and Preprocessing. Master's thesis, University of Florida, Gainesville, 1996.

[2]   V. Krishnaprasad, Event Detection for Supporting Active Capability in an OODBMS: Semantics, Architecture, and Implementation. Mater's thesis, University of Florida, Gainesville, 1994.

[3]   S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim, Composite Events for Active Databases: Semantics, Contexts, and Detection. In *Proceedings, International Conference on Very Large Data Bases*, pages 606-617, August 1994.

[4]   H. Liao, Global Events In Sentinel: Design and Implementaion of a Global Event Detector, University of Florida, Gainesville, 1997.

[5]   E. Anwar, L. Maugis, and S. Chakravarthy. A New Perspective on Rule Support for Object-Oriented Databases. In *Proceedings, International Conference on Management of Data*, pages 99-108, Washington, D.C., May 1993.

[6]   OODB. OpenOODB Toolkit, Release 0.2 (Alpha) Document. Texas Instruments, Dallas, September 1993.

[7]   Hung-ju Chu, A flxible ECA Dynamic Rule Editor for Sentinel: Design and Implementation. Master's thesis, University of Florida, Gainesville, 1998.

[8]   John Bloomer, Power Programming with RPC. O'Reilly & Associates, Inc, February 1992.

[9]   Nichols B, Buttlar D & Farell J.P, Pthreads Programming: A POSIX Standard for better Multiprocessing, O'Reilly publications, 1st Ed, September 1996.

[10]   Donald Lewine, Posix Threads programming guide. September 1994.

[11]   Jennifer Sung, A Recoverable Asynchronous Event Manager for Supporting distributed Active databases. Master's thesis, University of Florida, 1998.

[12]   S. Han, Three-Tire Architecture for Sentinel Applications and Tools: Separating Presentation from Functionality, University of Florida, Gainesville, 1997.

BIOGRAPHICAL SKETCH

Gauri Sukhatankar was born on October 1, 1973 in Mumbai, India. She received her Bachelor of Science degree in electrical and computer engineering from the University of Florida in December 1996. In the Spring of 1997, she started her graduate studies in computer and information science and engineering at the University of Florida. She expects to receive her Master of Science degree in computer and information science and engineering from the University of Florida, Gainesville, Florida, in May 1999. Her research interests include Active and Object-oriented databases and Data Warehousing.